ALGORITHMS FOR SELF-GROWING AND HIGHER-ORDER NEURAL NETWORKS

By

S Mohammad Saleem

DEPARTMENT OF ELECTRICAL ENGINEERING

Indian Institute of Technology Kanpur

JANUARY, 2002

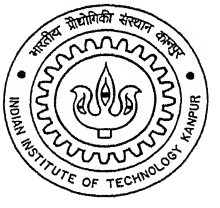
ALGORITHMS FOR SELF-GROWING AND HIGHER-ORDER NEURAL NETWORKS

A Thesis submitted
in partial fulfilment of the requirement
for the degree of

MASTER OF TECHNOLOGY

by

S Mohammad Saleem

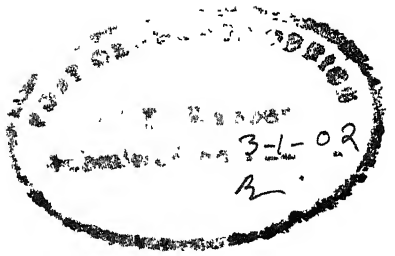


To the

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

January 2002



CERTIFICATE

This is to certify that the work contained in this thesis entitled **"Algorithms for Self-growing and Higher-order Neural Networks"**, by S Mohammad Saleem, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr P K Kalra
Professor
Department of Electrical Engineering
Indian Institute of Technology, Kanpur

ABSTRACT

The area of dynamically altering neural network architecture is studied. Cascade correlation algorithm and its variants are considered. An attempt has been made to investigate the problems posed by self-growing architectures and to improve the performance of these algorithms. Various methods of cascading architectures "cascor", "caserr", "casall" are developed.

The complexity of neuron in terms of enhanced inputs and their activations are studied. Both the static and dynamic methods of adding higher order complexities are investigated. Generalized neuron model and the different architectures that can be built with the generalized neuron as the basic node are developed The improved variant of cascade architecture with the best combination of available inputs, "casany" is developed. Various problems that hinder the learning process are dealt with suitable solutions. The algorithms developed are validated on several benchmark problems.

ACKNOWLEDGEMENT

I'd like to take the opportunity to express my sincere thanks to my thesis supervisor Dr. P K Kalra for affording me an ambience where I could openly discuss my problems and ideas. It was a great pleasure to work under him for I gained much through the interactions.

My stay at IITK has been very fruitful for not only did I gain knowledge but also friends with whom I shared my views, met teachers who influenced my thought, and benign experiences that enriched my self and contributed to an all-round growth and well being. I thank my parents and my brother Haneef Javed for all the support provided throughout, for their affections and encouragement.

I'd thank my batch-mates Sankar and Manoj for being with me all long as we did the courses together or shared a snack at Hall 4 canteen. My discussions with them gave me a feel for the algorithms I implemented for I could come up with certain schemes, which gave a direction of thought to the problems posed.

The atmosphere in the lab paved way for my betterment as I interacted with Madhav Krishna, Prashant, Maj. Jayesh, Maj. Vaid, Maj Mohan Kumar, Maj. Patil, Manu and others. My sincere thanks to them. I am also thankful to Ashesh, Balaji and others who added all the flavors of joy and made my stay at IITK wonderful.

I would like to express my gratitude to all those who directly or indirectly helped me through the successful completion of my work.

S Mohammad Saleem

CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Several Neural Network architectures and learning algorithms have been proposed for the application areas such as function approximation and classification tasks. The backpropogation algorithm offers a way to learn arbitrarily complex boundaries using multi-layer feed-forward networks. Such networks consist of one or more layers of hidden nodes between the input and output layers. The choice of the number of layers and the hidden nodes in each layer has a significant impact on the performance of networks. Thus one of the major criticisms of fully supervised feed-forward neural networks is their failure to cope with situations, which may require a novel topology. This is not so much due to the limitations of particular learning algorithms but is rather due to the limits of the networks structure and how these are set. The choice of network architecture and the type of neuron is usually a matter of trial and error.

1.1 Problem Statement

Neural network can be externally viewed as a 'black box'; all the internal topological issues can be altered by the learning algorithm. A neural network can have any number of inputs and any number of outputs. These are fixed by the characteristics of the problem. How many hidden nodes are required, what should be the node characteristics, how should the nodes be connected and how many layers the problem will require is not clear. Too small a network cannot learn the problem well, but too large a size may lead to over-fitting and poor generalization performance.

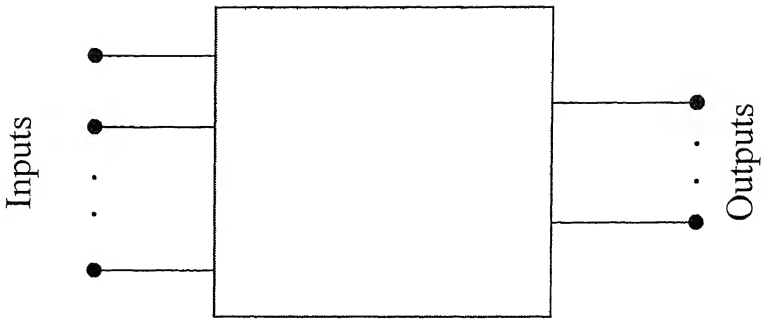


Fig 1.1 The conceptual external view of a neural network, ignoring all internal topology issues, a true "black box".

The study of dynamically altering network topologies to suit a particular problem during the training of fully supervised feed-forward neural networks is investigated. The concept of including higher order terms that enhances the node characteristics is also investigated. Dynamically altering networks presents good opportunities for developing optimal network architectures that generalize well.

1.2 Dynamic Learning Algorithms

There are different approaches that alter the network architecture as learning proceeds. Some of these types of algorithms are Constructive Algorithms (adding sections to the network) and Pruning Algorithms (removing sections from the network).

Constructive algorithms start with a small network and then add additional units and weights until a satisfactory solution is found. Pruning algorithms involves using a larger than needed network and training it until acceptable solution is found. After this, some hidden units or weights are removed where they are no longer actively used.

1.2.1 Constructive Methods Vs Pruning Methods

The constructive approach has a number of advantages over the pruning approach[1]. First, for constructive algorithms, it is straightforward to specify an initial network (only the output layer having as many neurons as the number of outputs specified by the problem), whereas for pruning algorithms, it is not known in advance how big the initial network should be. Second, constructive algorithms always search for small network solutions first. They are thus more computationally economical than pruning algorithms, in which the majority of the training time is spent on networks larger than necessary. Third, as many networks with different sizes may be capable of implementing acceptable solutions, constructive algorithms are likely to find smaller network solutions than pruning algorithms. Smaller networks are more efficient in forward computation and can be described by a simpler set of rules. Fourth, pruning algorithms usually measure the change in error when a hidden unit or weight in the network is removed. However, such changes can only be approximated for computational efficiency, and hence may introduce large errors, especially when many are to be pruned.

There are a number of inherent advantages in using constructive algorithms over algorithms that begin training with an oversize network. First, it is often difficult to specify what size network can actually be considered an oversize network *a priori*. If the initial network selected is too small, it will be unable to converge to a good solution and hence under fit the data. On the other hand, selecting an initial network that is much larger than required makes training computationally expensive. Constructive algorithms however, initially select a minimal size network and can increase the network complexity until an appropriate level is reached. In addition, constructive algorithms will spend the majority of

their time training networks smaller than the final network, as compared to algorithms that start training with an oversize network.

Another advantage of constructive algorithms is that the problem of encountering poorly performing local minima is avoided. The majority of training algorithms is based on gradient methods, and hence can become trapped in local minima. Constructive algorithms are able to escape from a local minimum when more weights are added to the network. This can occur because the addition of more weights increases the dimensionality of the error surface and may allow the network to continue reducing the error level.

1.3 Constructive Approach

There are number of properties that must be defined for constructive algorithms. These include the type of training algorithm, establishing how the new hidden neuron is connected to the current network, defining which weights are to be updated and in which order, deciding on criteria to determine when a new hidden neuron is added and deciding on criteria to halt network construction.

In general, constructive networks use gradient-based training algorithms due to their convergence speed. A number of ways of connecting a new hidden neuron to the network exist. The two common methods are to construct a single layer of hidden neurons or to create a cascade of hidden neurons. In cascade architecture, each new hidden neuron receives input connections from all the network inputs and previously installed hidden neurons. Since the hidden neurons in cascade architecture receive additional information from some non-linear combination of the inputs (implemented by previous hidden neurons), these neurons are termed higher order neurons and are capable of performing a

more complex function of the input variables. Cascade networks, while having more representational power, are more likely to overfit the data.

The ability to control the complexity of the new hidden neuron is an important issue for constructive networks in terms of convergence speed and generalization. A higher order neuron model, which is likely to possess better mapping and classification capability, can be considered as basic node. The concept of including higher order complexities and the internal architecture of a neuron is addressed in Chapter 5.

1.4 Organization of the thesis

The objective of this thesis is to investigate the area of dynamically altering networks and bring in best network topologies. Chapter 2 discusses the Cascade correlation algorithm and its improvements. Various methods of cascading architectures "Caserr" and "Casall" are considered in Chapter 3. The complexity of the neuron in terms of enhanced inputs and their best activations are studied in Chapter 4. Both the static and dynamic methods of adding higher order complexities are investigated. Chapter 5 addresses the "Casany" algorithm that includes the best generalizations during the build up process. Various problems that hinder the learning process are dealt with appropriate solutions. The algorithms developed are validated on several benchmark problems and the results are discussed in Chapter 6.

CHAPTER 2

CASCADE CORRELATION

One of the Constructive Algorithms is the Cascade-Correlation algorithm. Instead of just adjusting the weights in a network of fixed topology, Cascade-Correlation begins with a minimal network, then automatically trains and adds new hidden units one by one, creating a multi-layer structure [2]. Once a new hidden unit has been added to the network, its input side weights are frozen. This unit then becomes a permanent feature-detector in the network, available for producing outputs or for creating other more complex feature-detectors.

2.1 Description of Cascade-Correlation

In cascade correlation architecture, hidden units are added to the network one at a time and do not change after they have been added. The learning algorithm, which creates and installs the new hidden units, is correlation based, which maximizes the magnitude of the correlation between the new unit's output and the residual error signal.

The cascade architecture begins with the given number of inputs and one or more output units, but with no hidden units. The number of inputs and outputs is dictated by the problem. Every input is connected to every output unit by a connection with an adjustable weight. There is also a bias input, which is set.

Hidden units are added to the network one by one. Each new hidden unit receives a connection from each of the network's original inputs and from every pre-existing hidden

unit. The hidden unit's input weights are frozen at the time the unit is added to the net; only the output connections are trained repeatedly Each new unit therefore adds a new one-unit layer to the network. This leads to the creation of very powerful high-order feature detectors; it also may lead to very deep networks and high fan-in to the hidden units.

The learning algorithm begins with no hidden units. The direct input-output connections are trained as well as possible over the entire training set. At some point, this training will approach an asymptote. When no significant error reduction has occurred after a certain number of training cycles (controlled by a "patience" parameters set by the user), the network is run one last time over the entire training set to measure the error If the network's performance is within tolerable limits the training is stopped, if not, a new hidden unit is added to the network. To create a new hidden unit, a candidate unit that receives trainable input connections from all of the networks external inputs and from all pre-existing hidden units is selected. The output of this candidate unit is not yet connected to the active network. The training set is presented over a number a passes adjusting the candidate unit's input weights after each pass, while all other weights are unaltered The goal of this adjustment is to maximize the Correlation between the candidates' output and the networks residual error. The Correlation measure [2] is calculated as follows.

$$C = \sum_{O} \left| \sum_{P} \left(Y_P - \overline{Y} \right) \left(E_{p,O} - \overline{E_O} \right) \right| \tag{2.1}$$

where, P is the training pattern, o is the network output at which the error is measured. Y_P is the candidates' output for the pattern P, $E_{P,O}$ is the networks error at output o, for the pattern P. $\overline{Y}$ and $\overline{E_O}$ are the respective values of Y and E_O averaged over all the patterns.

The Correlation measure is maximized performing gradient accent rule. The partial derivative of C with respect to candidate units input weights, $\partial C/\partial w_i \ \forall w_{i=all \text{ input connections}}$ is obtained as

$$\frac{\partial C}{\partial w_i} = \sum_{P,O} \sigma_O \left(E_{P,O} - \overline{E_O}\right) fnet'_P \ X_{i,P} \qquad (2.2)$$

where σ_O is the sign of the correlation of the candidate's value and output o, $fnet'_P$ is the derivative for pattern P of the candidate unit's activation function with respect to the net_P (sum of its inputs), and $X_{i,P}$ is the input to the candidate unit receives from unit i (network input or previous hidden unit's output) for pattern P.

Only single layers of weights are trained at a time by performing gradient ascent. The Correlation measure C is maximized to a desired value, or until C stops improving. Then this new unit is installed in the active network, by freezing all its input weights. The output connections are trained until the network's residual error reduces to a desired value, or saturates (no further significant reduction in error). A new candidate unit is considered for the later case and the cycle is continued.

In the equation (2.1), absolute value of correlation measure is considered, i.e., a candidate unit cares only about the magnitude of its correlation with the error at a given output, and the sign of the correlation is considered in (2.2), in maximizing the correlation If a hidden unit correlates positively with the error at a given output unit, a negative connection weight is developed to that output unit, thus some of the error is cancelled. On the other hand, if the correlation is negative, the output weight is positive. Since a unit's weights to different outputs may be of mixed sign, a unit can sometimes serve two

purposes by developing a positive correlation with the error at one output and a negative correlation with the error at another.

The learning algorithm used here is simple back-propagation with momentum. All the weights are randomly initialized. The error function used is sum-square error. The patience parameters used are maximum number of hidden units that are to be installed, the permissible residual error, maximum correlation measure, maximum number of iterations for maximizing the correlation measure, maximum number of iterations for output connections training (active network training) and the size of each candidate pool.

The base case is no hidden nodes at all, thus forming just a perception style output layer. The learning process stops when the error generated is less than the permissible residual error, or the when the number of hidden units installed reaches the maximum number of hidden units indicated. To further improve the training process in the later case, the partially trained network can be reloaded with the upper limit of hidden units increased. Thus, the training process not only starts with the base case (only output layer), but with partial network also.

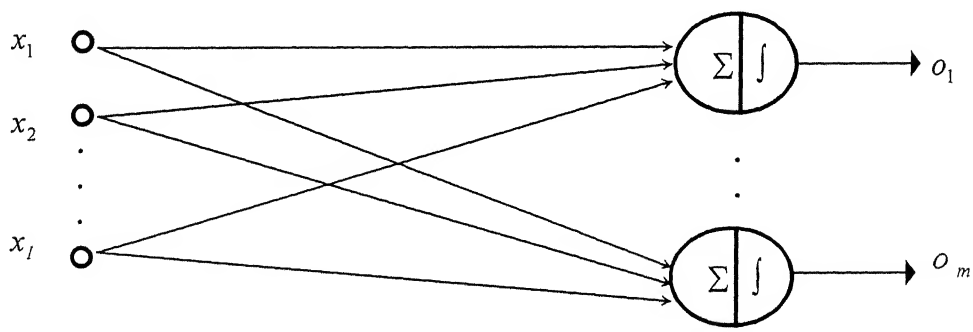


Fig 2.1 Base Case: With no hidden units All inputs are connected to all output nodes.

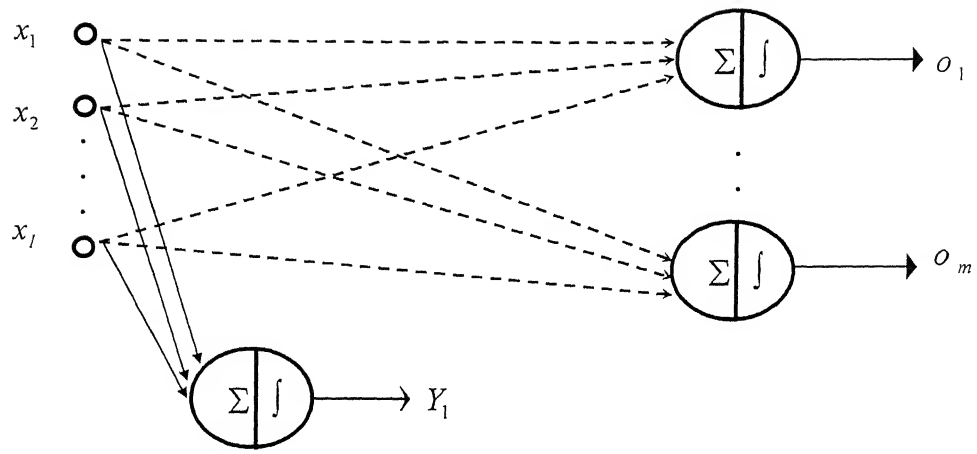


Fig 2.2 Illustrates addition of hidden unit 1
Dotted connections indicate inactive connections. Solid lines indicate active connections.

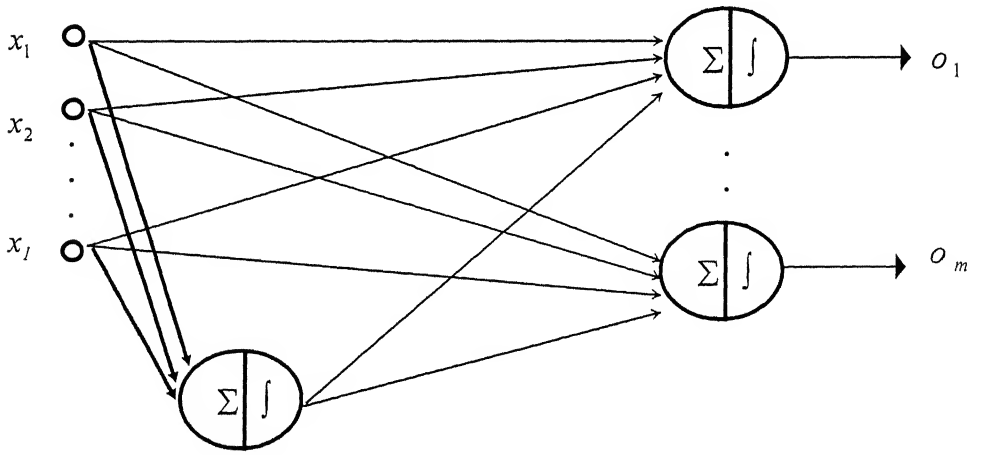


Fig 2 3 Illustrates addition of hidden unit 1 (Input weights freezing)
Thick solid lines indicate frozen connections. Thin solid indicate active connections

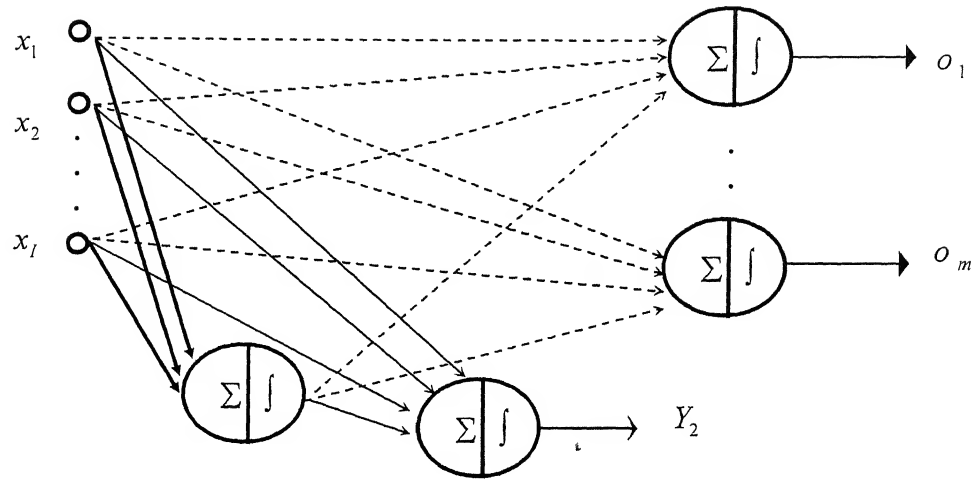


Fig 2 3 Illustrates addition of hidden unit 2.
Hidden unit 2 receives connections from all the inputs and from hidden unit 1 Dotted connections indicate inactive connections. Solid lines indicate active connections. Thick solid lines indicate frozen connections.

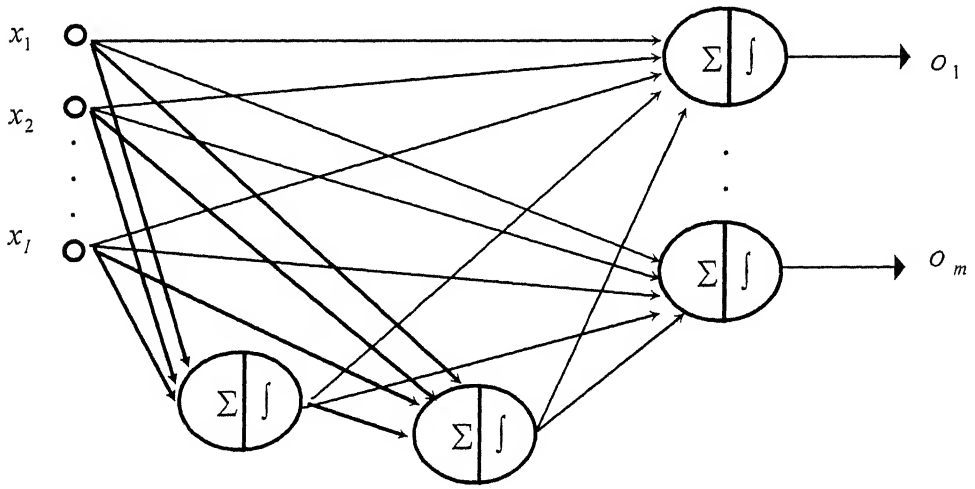


Fig 2.5 Cascade architecture with two hidden units
Thick solid lines indicate frozen connections. Thin solid indicate active connections

2.2 Improvements

2.2.1 Correlation Measure Modifications

The Correlation measure given by equation (2.1) is the correlation coefficient between the residual error and the new hidden unit activation. The correlation coefficient determines the extent to which values of two variables (network residual error and the new unit's activation) are "proportional" to each other. Proportional means linearly related, that is, the correlation is high if it can be approximated by a straight line (sloped upwards or downwards). This line is called the regression line or least squares line, because it is determined such that the sum of the squared distances of all the data points from the line is the lowest possible. The correlation coefficient given by equation (2.1), takes into account the summation term over all patterns. This measure increases with the number of patterns and with the number of outputs. The choice of patience parameter 'maximum correlation measure' is problem dependent. Instead, Pearson Correlation, which assumes that the two variables are measured on at least interval scales, can be used. It is given as

$$P.C = \frac{1}{O} \frac{\sum_{O} \left| \sum_{P} \left(Y_P - \overline{Y}\right)\left(E_{P,O} - \overline{E_O}\right) \right|}{\sqrt{\left[\sum_{P} \left(Y_P - \overline{Y}\right)^2 * \sum \left(E_{P,O} - \overline{E_O}\right)^2 \right]}} \qquad (2.3)$$

The value of Pearson Correlation lies between 0 and 1. The maximum Correlation measure in this case is set to a value which nearer to 1.

2.2.2 Pool of Candidates

Once a new hidden unit has been added to the network, its input-side weights are frozen. This unit then becomes permanent feature detector in the network, available for producing outputs or for creating other, more complex feature detectors. Therefore instead of a single candidate unit, a pool of candidate units, each with a different set of random initial weights, are considered [3]. All these candidate units receive the same inputs and try to reduce the same residual error, without any interaction among them. The candidate whose correlation is the best is the winner, and is installed in the active network. Selecting the best unit from a pool of units greatly reduces the chance that a less useful unit will be permanently installed and as many parts of the weight space can be explored, it can be ensured that good candidate unit will be finally installed.

The size of the candidate pool and the weight space explored by each unit in the pool are selected in such a way that all the candidates explore different weight spaces; or the weight space of one unit forms superset of the other beginning with a small space explored by the first unit, which forms subset explored by the second unit, and the last unit in the pool explores maximum weight space. It is found that the latter case is better.

2.3. Termination Criteria

The training phase of a candidate unit terminates in three cases; when the correlation coefficient reaches the maximum correlation measure, or when there is no improvement in the correlation value (either saturates or the change of value is significantly low) or the number of new unit's training phase iterations reaches the maximum specified limit. The unit, which correlates best, is selected from the pool for the

active network, by connecting this unit to the output layer units The minimum limit on the correlation measure can also be specified If none of the candidate units' correlation with the residual error crosses this limit, then an entirely new pool is considered.

The active network is then trained with the latest configuration. This halts when the error reaches below the specified value, or when there is no further reduction in the error value (or stagnates), or the number of iterations reaches the maximum specified limit.

2.4 Cascade Correlation Algorithm (CasCor)

The precise Cascade Correlation algorithm is as follows

Create a network without hidden units

DO (repeat)

Train Output Connections

IF *the network error is not reduced below the specified tolerance or the network saturates*

THEN *Create and Train Candidate units*

Insert Best Candidate unit into the network and its input weights are frozen

END

WHILE *End of training*

Create and Train Candidate units:

Consider a pool of size 'm'

Create 'm' candidate units C_1 to C_m

Connect their inputs with the output of the input units and all previously existing hidden units

DO (repeat)

Perform a gradient ascent step for correlation measure maximization of candidates' activation with output deviation

WHILE *End of Candidate training*

Insert Best Candidate unit into the network:

Select Candidate with highest covariance

Delete all other Candidates

Connect the selected unit to the output units

Train Output Connections:

DO (repeat)

Perform gradient descent step for output error on the output weights

WHILE *End of output training*

2.4.1 Advantages

Cascade-Correlation algorithm has the following advantages:

The size, depth, and connectivity pattern of the network need not be guessed in advance. A reasonably small (though not optimal) net is built automatically.

Cascade-Correlation learns fast. In back prop, the hidden units engage in a complex dance before they settle into distinct useful roles; in Cascade-Correlation, each unit sees a fixed problem and can move decisively to solve that problem.

At any given time, only one layer of weights in the network is trained. The rest of the network is not changing, so results can be cached.

The error signals are not propagated backwards throughout the network connections. A single residual error signal can be broadcast to all the pool candidates. The weighted connections transmit signals in only one direction, eliminating one troublesome difference between back prop connections and biological synapses.

Cascade-Correlation can build deep nets without the dramatic slowdown as compared to back-propagation networks with more than one or two hidden layers.

Cascade-Correlation is useful for incremental learning, in which new information is added to an already-trained net. Once a feature detector is added, it is available from that time for producing outputs or more complex features.

The candidate units in a pool do not interact with one another, except to pick a winner Each candidate sees the same inputs and error signals at the same time. This limited communication makes the architecture attractive for parallel implementation.

Even if the specified maximum network size will not be able to reduce the error completely, the size of the network can always be further increased and by loading the

partial trained network, the training process will be continued until the desired results are obtained.

2.4.2 Limitations

In cascade architecture, hidden units are added to the network one at a time and do not change after they have been added. This is because of the input layer weight freezing. When a bad unit comes into the architecture, it increases the network architecture for compensation.

When a new unit is to be selected, the correlation between the residual error and the new hidden unit activation is determined and then the correlation measure is maximized. For a pool of candidate units, the correlation of each unit's activation with the residual error has to be maximized. As the size of the pool increases, the time complexity increases. If the size of the pool is small, the weight space explored may not be good enough, and it may lead to selection of poorly correlated or wrong unit.

To create a new hidden unit, a candidate unit receives trainable input connections from all of the network's external inputs and from all pre-existing hidden units. As the number of hidden units increases, the fan-in to hidden units also increases, leading to complex architecture.

CHAPTER 3

CASERR AND CASALL

The Cascade Correlation algorithm determines and maximizes the Correlation measure between the residual error and the new hidden unit activation. The correlation coefficient determines the extent to which network residual error and the new unit's activation are "proportional" to each other. Proportional means linearly related; that is, the correlation is high if it can be approximated by a straight line (sloped upwards or downwards). This line is called the regression line or least squares line, because it is determined such that the sum of the squared distances of all the data points from the line is the lowest possible. The input connections of the new hidden unit are modified to maximize the correlation. When this new hidden unit is brought in the active network, its output connections to the output layer neurons are established. These connections are initialized, and the output layer weights are trained. Two factors will affect the training process, the new connections in the output layer and the correlation measure. The new connections if not correctly initialized, the training length increases. The sign of these connections is the inverse of the covariance with the respective output unit.

3.1 Description of CasErr

In principle, the correlation measure may not be always a well-suited target function for training the new candidate units. Maximum correlation trains candidates to have a large activation i.e., large deviation from average activation, whenever the error at

their output is not equal to the average error. When the error deviation is only small, a large activation leads to higher payoff than a small one, even though the latter would be appropriate. Therefore, the cascade correlation algorithm has a tendency to overcompensate errors.

The learning rule can be modified and train the candidate units directly for minimization of output errors instead of for maximization of correlation [4]. The modifications required are creating virtual output connections for the candidate units. The training process can be incremental i.e., only the new unit considered is trained, or the network with the virtual connections and the output layer connections is trained. In the incremental training only the input connections of the candidate unit and its output connections to the output layer units are modified, while the other connections are held constant. In the later case, the weights of the network without the candidate unit connections i.e., partially trained network's weights are stored before they are subjected to train. Then the virtual connections along with the output layer connections are trained by the back propagation algorithms. In both type of training process, the new connections are accepted if the network's error reduces. Otherwise, another candidate unit is considered, by retaining the partially trained network. In this algorithm also, once the hidden unit is inserted, its input connections are frozen. In addition, a pool of candidate units that are initialized in different weight regions or different sub-regions are considered.

Modifying both the output layer weights and virtual connections (two-step training) simultaneously offers advantage over incremental training (three-step training). In the incremental training process, the partially trained network rests and watches which of the candidate unit from the pool reduces much of the error, then the best unit is brought into

the network with the trained weights and the entire network is made active and further trained. In the other case, the partially trained network pulls up the new units along with it and looks for the best unit that reduces the network error, on a run. Whenever a new unit is to be added, the partially trained network weights are stored. Each candidate unit in a pool starts with these weights as the initial network plus their weights, and the training progresses for specified iterations The unit, which has reduced error efficiently, is the winner, its virtual connections are set into the network and the entire network is further trained.

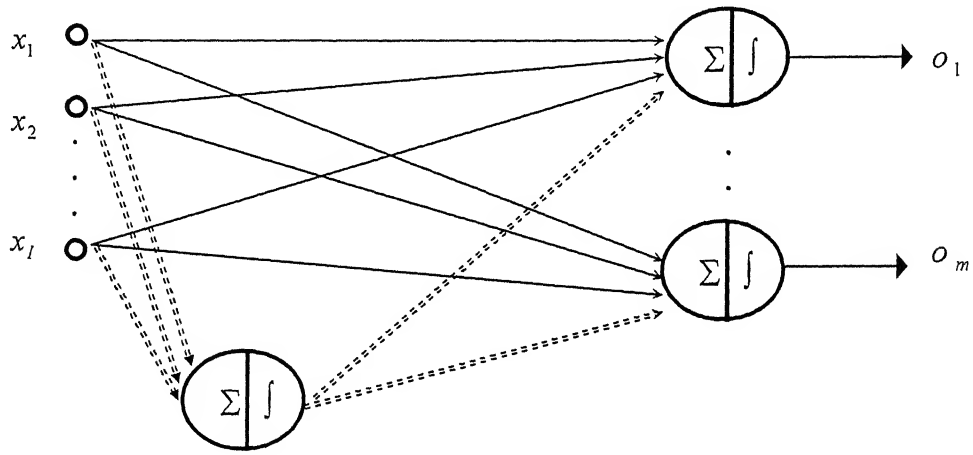


Fig 3.1 CasErr: Addition of hidden unit 1.
Solid lines indicate active connections. Double dotted lines indicate virtual connections

3.2 CasErr Algorithm

The precise Cascade Error minimization algorithm is as follows

Create a network without hidden units

DO (repeat)

Train Output Connections

IF *the network error is not reduced below the specified tolerance or the network saturates*

THEN ***Create and Train Candidate units***

Insert Best Candidate unit into the network and its input weights are frozen

END

WHILE *End of training*

Create and Train Candidate units:

Consider a pool of size 'm'

Create 'm' candidate units C_1 to C_m

Connect their inputs with the output of the input units and all previously existing hidden units. Their outputs are connected with the output layer units. All these connections are *virtual connections*.

DO (repeat)

Perform gradient descent step for output error on the output weights and on the virtual connections.

WHILE *End of Candidate training*

Insert Best Candidate unit into the network:

Select Candidate with highest error reduction

Delete all other Candidates

The selected unit's virtual connections are set into the network

Train Output Connections:

DO (repeat)

Perform gradient descent step for output error on the output weights

WHILE *End of output training*

3.3 Description of CasAll

Many heuristics have been used to guide the search in the possible solution space. Hidden units are added to the network one by one. Each new hidden unit receives a connection from each of the network's original inputs and also from every pre-existing hidden unit. The hidden unit's input weights are frozen at the time the unit is added to the net; only the output connections are trained repeatedly. It is found that the impact of freezing on the number of training epochs, percentage of correctness on the test set, and number of hidden units cascaded is different for different problem domains and for different weight initializations.

Because of this input weight freezing, the next hidden units have to correct errors (if any) caused by the preceding hidden units Also as the number of hidden units increases, the fan-in to the next hidden units also increases, leading to complex architecture Instead of adding only a hidden unit which receives inputs from all existing nodes, the architecture can be modified such that hidden layers are added [5], in such a way that the units in each hidden layer receives inputs only from the preceding layer units. In each layer, the units are added one at a time. With this architecture, there is no need of input weight freezing and all the connection values can be changed as the training process progresses.

The architecture, which results when two hidden layers with four units each are specified, is shown in figure 3.2.

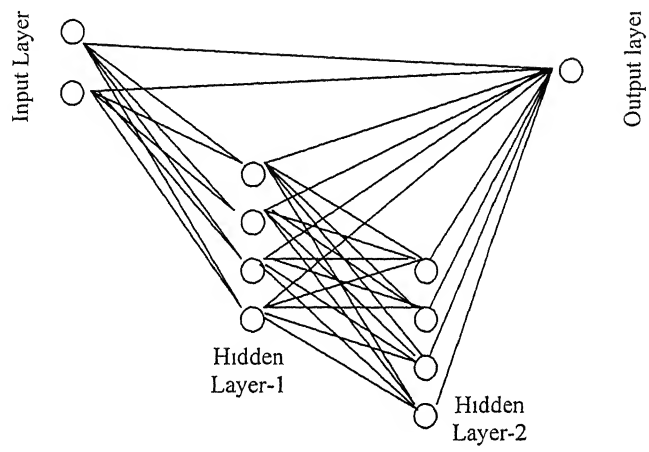


Fig 3.2 CasAll architecture – 2 hidden layers with 4 units each

3.4 CasAll Algorithm

The precise CasAll algorithm is as follows

Create a network without hidden units

DO (repeat)

Train all existing Connections

IF *the network error is not reduced below the specified tolerance or the network saturates*

THEN ***Create and Train Candidate units***

Insert Best Candidate unit into the network

END

WHILE *End of training*

Create and Train Candidate units:

Consider new hidden layer if there are no hidden layers or the existing hidden layers are full to their specified capacity.

Consider a pool of size 'm'

Create 'm' candidate units C_1 to C_m

Connect their inputs with the output units in the preceding layer.

If Error minimization is considered:

The output of the new hidden unit is connected with the output layer units. All these connections are *virtual connections*.

DO (repeat)

If Correlation measure is considered:

Perform a gradient ascent step for correlation measure maximization of candidates' activation with output deviation

If Error minimization is considered:

Perform gradient descent step for output error on all weights.

WHILE *End of Candidate training*

Insert Best Candidate unit into the network:

Select Candidate with highest error reduction or Select Candidate with highest covariance depending on the algorithm choice.

Delete all other Candidates

The selected unit's virtual connections are set into the network/ Connect the selected unit to the output units depending on the algorithm choice.

Train Output Connections:

DO (repeat)

Perform gradient descent step for output error on all weights

WHILE *End of output training*

"CasAll" algorithm gives layered architecture, thereby limiting the fan-in to the hidden units. In addition, the structure is free from weight freezing, allowing the units to correct the errors by themselves rather than leaving the whole job for units to be added.

CHAPTER 4

GENERALIZED NEURON

The Standard Neuron has combination of aggregation and activation functions. The standard form of aggregation function can be linear weighted sum (linear basis function), and the most common activation functions are sigmoidal or tangent hyperbolic functions. Such neural models when used to solve real life problems may require a large number of neurons in standard neural network (STD). It is also well known that the number of unknowns to be determined to fix the architecture of the artificial neural network (ANN) grows with the number of neurons and the hidden layers. Therefore, the working of ANN becomes computation and memory intensive. The computational burden can be reduced either by reducing the number of neurons or by improving the learning techniques The number of neurons in the ANN in turn depends on the neuron model itself. A higher neuron model may produce better ANN with fewer neurons.

4.1 Higher Order Neurons

The standard neuron model is a first order model. This has limitations of mapping and classification. These limitations can be overcome either by using second order gradient techniques or by using higher order neuron models. The second order gradient techniques, such as conjugate gradient and quasi-Newton methods, instead of simple gradient technique achieve rapid convergence near a minimum. Second order gradient techniques may reduce the architectural complexity but not the learning

complexity. These methods, moreover, require more storage capacity, thus the working of these methods becomes computation and memory intensive.

As the mapping and the classification power of a neuron depend on its order, a higher neuron model is likely to possess better mapping and classification capability. Higher order neuron model, which includes quadratic and higher order basis functions in addition to linear basis function reduce the learning complexity. The architectural complexity increases with the number of higher order basis functions The overall complexity of working of the ANN with higher order neurons is less compared to second order gradient techniques.

4.2 Generalized Neuron Model

A generalized neuron (GN) model is considered which has 'N' aggregation functions and 'F' activation functions. The aggregation functions can be linear weighted sum (linear basis function), quadratic and higher order basis functions. The activation functions can be linear or various types of non-linear functions.

The cross-products of the input terms can be added into the model, where each component of the input pattern multiplies the entire input pattern vector. A reasonable way to do this is to add all interaction terms between input values. For example, for a back-propagation network with three inputs (A, B and C), the cross-products would include: AA, BB, CC, AB, AC, and BC. This example adds second-order terms to the input structure of the network.

Additional input nodes can be functional expansion of the base inputs. Thus, a back-propagation model with A, B and C might be transformed into a higher-order neural network model with inputs: A, B, C, f(A,B,C),g1(A,B),g2(B,C) etc. In this model, input variables are individually acted upon by appropriate functions. Many

different functions can be used. The overall effect is to provide the network with an enhanced representation of the input No new information is added, but the representation of the inputs is enhanced. Higher-order representation of the input data can make the network easier to train. The joint or functional activations become directly available to the model.

Generalized neuron with three aggregations and two activations is shown in fig 4.1.

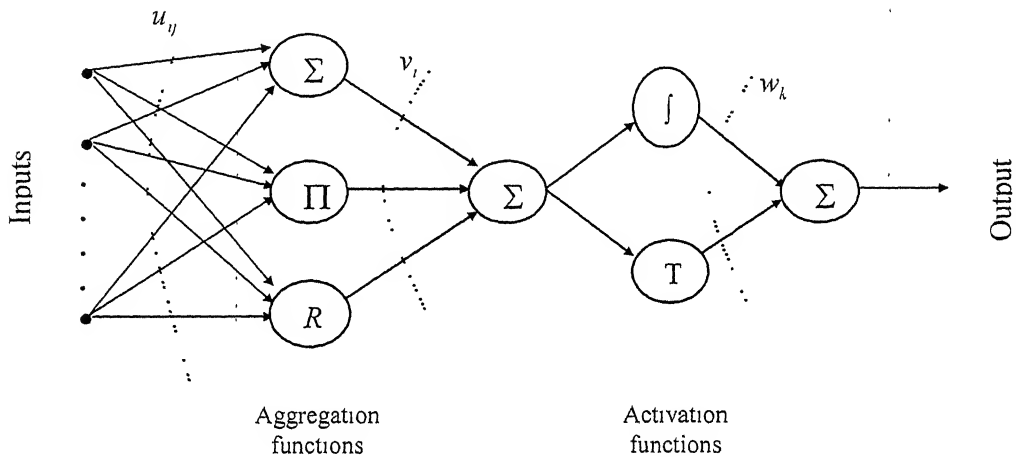


Fig 4.1 Generalized Neuron (3 Aggregation functions, 2 Activation functions)

Inputs: $x_1, \ldots\ldots, x_I$

Aggregation functions:

Σ : Weighted Sum of inputs, $\sum_{i=1}^{I} u_{ij} x_i$, here $j = 1$

Π : Sum of products of two weighted inputs (SOP2), $\Sigma(\prod_{2} u_{ij} * x_j)$, here $j = 2$

R : Radial Basis term (RB), $\frac{1}{\sqrt{\sum_{i=1}^{I}(x_i - u_{ij})}}$, here $j = 3$

Activation functions:

∫ : Sigmoidal activation function

T : Tan hyperbolic activation function

In some cases, a hidden layer is no longer needed, when generalized neuron is the basic neuron. However, there are limitations to the network model. Many more input nodes must be processed to use the transformations of the original inputs. With higher-order systems, the problem is exacerbated. Yet, because of the limitations imposed on finite processing time in some cases, it is important that the inputs are not expanded more than is needed to get an accurate solution.

4.3 GN Network Architecture

The generalized neuron model ANN can be homogeneous or heterogeneous Homogeneous network has same architectural neurons (same number of aggregation functions and activation functions for all the neurons). In Heterogeneous network, the neurons have different configurations. Heterogeneous network can be semi-heterogeneous, where each layer has same architectural neurons, different from other layer neurons.

Sigma-Pi-Sigma model (SPS) is a semi-heterogeneous network with three layers. In the first layer of the SPS, only the summation is done as in STD architecture. Output of this layer is multiplied two at a time and summed up which serves as the input to the next hidden layer neurons. The final layer, which is the output layer of SPS, receives inputs from the preceding layer neurons, where the summation is done.

Various architectures can be explored with generalized neuron as a basic node.

4.4 Mathematical Analysis of GN Network Architecture

Consider 'N' aggregation functions and 'F' activation functions.

Let $n_1, n_2, \ldots, n_N$ denote aggregation functions 1,2, ..N.

$f_1, f_2, \ldots, f_F$ denote activation functions 1,2,...F.

sn : weighted sum of $n_1, n_2, \ldots$ and n_N

sf : weighted sum of $f_1, f_2, \ldots$ and f_F

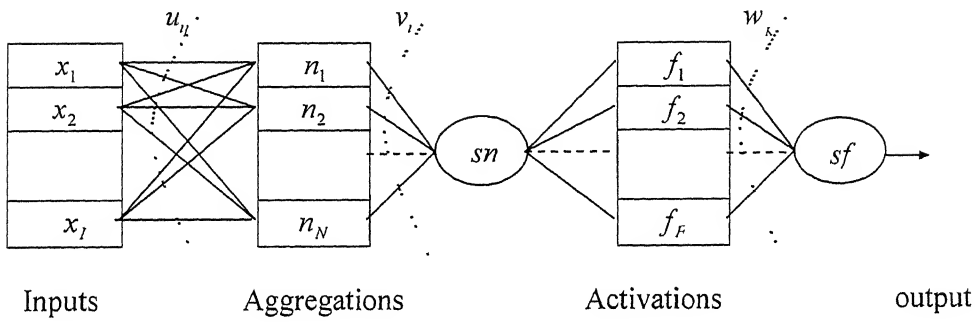


Fig 4.2 GN-Intra Connections

u_{ij} : Connection value between i^{th} aggregation and j^{th} input.

v_i : Connection value between i^{th} aggregation and sn

w_k : Connection value between k^{th} activation and sf

X · Input Vector of x_j

x_j : j^{th} input.

where j : 1,....I

i : 1,....N

k : 1,....F

$n_i : fn_i(u_{ij}, x_j) \quad \forall \; j : 1,. ..I$

$i : 1,....N$

fn_i : i^{th} aggregation function

$sn : \sum_{i=1}^{N} v_i n_i$

$f_k : ff_k(sn)$

ff_k : k^{th} activation function

$sf : \sum_{k=1}^{F} w_k f_k$

For the nodes in the first layer, input vector X, is the vector of problem specified inputs For all other layers, it is the vector of outputs from the preceding layer nodes.

Error value is computed (for sum-square error)

$E = \sum_{0=1}^{O} \frac{1}{2}(d_o - y_o)^2$ where O is the number of outputs.

The error signal vectors are computed as follows

For the nodes at the output layer l of O neurons (h =1,...O)

$$\frac{\partial E}{\partial w_k^{lh}} = \frac{\partial E}{\partial y_o} \frac{\partial y_o}{\partial w_k^{lh}} \qquad \forall \; k = 1,.....F$$

$$= (-1)(d_o - y_o) f_k^{lh}$$

$$= \delta_o^{lh} f_k^{lh} \quad , \text{ where } \delta_o^{lh} = (-1)(d_o - y_o)$$

$$\frac{\partial E}{\partial v_i^{lh}} = \frac{\partial E}{\partial y_o} \left[\sum_{k=1}^{F} \left(\frac{\partial y_o}{\partial f_k^{lh}} \frac{\partial f_k^{lh}}{\partial sn^{lh}} \right) \right] \frac{\partial sn^{lh}}{\partial v_i^{lh}} \qquad \forall \; i = 1,....N$$

$$= (-1)(d_o - y_o)\left[\sum_{k=1}^{F}\left(f_k^{lh\,'} w_k\right)\right] n_i \,, \qquad \text{where } f_k^{lh\,'} = \frac{\partial f_k^{lh}}{\partial sn}$$

$$= \delta_f^{lh} n_i \qquad \text{, where } \delta_f^{lh} = \delta_o^{lh}\left[\sum_{k=1}^{F}\left(f_k^{lh\,'} w_k\right)\right]$$

$$\frac{\partial E}{\partial u_{ij}^{lh}} = \frac{\partial E}{\partial y_o}\left[\sum_{k=1}^{F}\left(\frac{\partial y_o}{\partial f_k^{lh}}\frac{\partial f_k^{lh}}{\partial sn^{lh}}\right)\right]\frac{\partial sn^{lh}}{\partial n_i}\frac{\partial n_i}{\partial u_{ij}^{lh}} \qquad \forall\, i = 1,\ \ldots N \quad \forall\, j = 1,\ \ldots . I$$

$$= (-1)(d_o - y_o)\left[\sum_{k=1}^{F}\left(f_k^{lh\,'} w_k\right)\right] v_i^{lh} n_i^{lh\,'} \,, \quad \text{where } n_i^{lh\,'} = \frac{\partial n_i}{\partial u_{ij}^{lh}}$$

$$= \delta_n^{lh} n_i^{lh\,'} \qquad \text{, where } \delta_n^{lh} = \delta_f^{lh} v_i^{lh}$$

For the nodes in the layer, preceding to output layer

$$\frac{\partial E}{\partial w_k^{lh}} = \sum_{o=1}^{O}\left\{\frac{\partial E}{\partial y_o}\left[\sum_{k=1}^{F_0}\left(\frac{\partial y_o}{\partial f_k^{(l+1)o}}\frac{\partial f_k^{(l+1)o}}{\partial sn^{(l+1)o}}\right)\right]\left[\sum_{i=1}^{N_0}\left(\frac{\partial sn^{(l+1)o}}{\partial n_i^{(l+1)o}}\frac{\partial n_i^{(l+1)o}}{\partial y^{lh}}\right)\right]\right\}\frac{\partial y^{lh}}{\partial w_k^{lh}}$$

$$= \delta_o^{lh} f_k^{lh}$$

$$\text{where } \delta_o^{lh} = \sum_{o=1}^{O}\left(\delta_n^{(l+1)o} n_i^{(l+1)o\,'}\right) \quad \text{where } n_i^{(l+1)o\,'} = \frac{\partial n_i^{(l+1)o}}{\partial y^{lh}}$$

Thus, for all other nodes (h) in layers (l), the error signals are calculated as follows.

$$\frac{\partial E}{\partial w_k^{lh}} = \delta_o^{lh} f_k^{lh} \,, \qquad \forall\, k = 1,\ldots.. F^{lh}$$

$$\text{where, } \delta_o^{lh} = \sum_{nu=1}^{cnu(l+1)}\left(\delta_n^{(l+1)nu} n_i^{(l+1)nu\,'}\right)$$

nu = neuron number, $cnu(l+1)$: gives number of neurons in $(l+1)^{\text{th}}$ layer

$$\frac{\partial E}{\partial v_i^{lh}} = \delta_f^{lh} n_i, \qquad \forall\, i = 1,....N^{lh}$$

where $\delta_f^{lh} = \delta_o^{lh} \left[\sum_{k=1}^{F^{lh}} \left(f_k^{lh'} w_k \right) \right]$

$$\frac{\partial E}{\partial w_k^{lh}} = \delta_n^{lh} n_i^{lh'} \qquad \forall\, i = 1, ...N^{lh}, \quad \forall\, J = 1,.....I^{lh}$$

where $\delta_n^{lh} = \delta_f^{lh} v_i^{lh}$

N^{lh}, F^{lh}, I^{lh} are the corresponding values of N , F , I for l^{th} layer, h^{th} neuron.

The error gradients are thus computed and the weights are updated depending upon the algorithm.

The GN architecture variations are tested on various benchmark problems and the results are discussed in Chapter 6.

CHAPTER 5

IMPROVEMENTS TO CASCADE ARCHITECTURE AND ALGORITHMS

The generalized neuron has the capability of enhancing inputs, and hence fewer generalized neurons are required compared to standard neurons. In cascading architecture, these neurons can be considered as basic nodes that are added one by one. Further, not all the enhanced inputs may be required at a time. The generalized neuron can have 'N' aggregation functions and 'F' activation functions. The sums of the aggregations go through all the activations. Not all the interconnections in a neuron may be required at a time. The best combination of 'N' aggregations and 'F' activations forms a new node to be cascaded. A pool of candidate units, of all the possible combinations of aggregations and activations is considered. The candidate unit that best suits in the network is cascaded for further training.

The number of combinations of 'N' aggregations and 'F' activations are (2^N X 2^F). Each candidate in the pool has different aggregations and activations, thus each unit has different internal topology.

5.1 Selection of best topology (CasAny)

The possible combinations of 'N' aggregations are as shown in Table 5.1. The aggregations, which are set to '1', are active, otherwise inactive. There are 2^N

aggregations. Similarly, the possible combinations (2^F) of 'F' activations are as shown in Table 5.2.

Aggregations	*1*	*2*	. . . .	*N-1*	*N*
Possible Combinations	0	0	. . . .	0	1
	0	0	. . . .	1	0
	0	0	. . . .	1	1
	.	.	. . . .	.	.
	.	.	. . . .	.	.
	.	.	. . . .	.	.
	1	1	. . . .	1	1

Table 5.1 Aggregation functions-combinations

Activations	*1*	*2*	. . . .	*F-1*	*F*
Possible Combinations	0	0	. . . .	0	1
	0	0	. . . .	1	0
	0	0	. . . .	1	1
	.	.	. . . .	.	.
	.	.	. . . .	.	.
	.	.	. . . .	.	.
	1	1	. . . .	1	1

Table 5.2 Activation functions-combinations

The precise CasAny algorithm is as follows

Create a network without hidden units

DO (repeat)

Train all existing Connections

IF *the network error is not reduced below the specified tolerance or the network saturates*

THEN ***Create and Train Candidate units***

Insert Best Candidate unit into the network

END

WHILE *End of training*

Create and Train Candidate units:

Consider new hidden layer if there are no hidden layers or the existing hidden layers are full to their specified capacity.

Consider a pool of size m = $(2^N \text{ X } 2^F)$.

Create 'm' candidate units C_1 to C_m

Connect their inputs with the output units in the preceding layer.

If Error minimization is considered:

The output of the new hidden unit is connected with the output layer units. All these connections are *virtual connections*.

DO (repeat)

If Correlation measure is considered:

Perform a gradient ascent step for correlation measure maximization of candidates' activation with output deviation

If Error minimization is considered:

Perform gradient descent step for output error on all weights.

WHILE *End of Candidate training*

In cascade networks, the hidden neurons become more powerful higher order neurons as the network grows. Having neurons of too little complexity may slow convergence, while having neurons of too high complexity can cause poor generalization.

5.2 Heuristics in learning process

Whenever the present network is not capable of solving the problem, the training process reaches a plateau or the network error oscillates. One of the criteria for adding another neuron is met when the network error stagnates. Sometimes the network reaches local minima (which are not acceptable) between network error oscillations. When one of the criteria is met for considering a new neuron, the network may have moved in the maximum error direction because of the oscillations, thus introducing the chances of adding a wrongly learnt unit.

The heuristics followed in such cases is to keep track of local minima. Whenever the network error reaches a local minimum, the weights of the network corresponding to the local minimum are stored temporarily. The network is allowed to learn further, and if the network error reduces, the stored temporary weights are ignored. In the other case if the network oscillates or moves through, various local minima, the network weights

corresponding to the least network error are stored. Whenever the criteria is met for considering a new neuron, the network weights corresponding to the least error are restored, and further training is continued from this point.

The partially trained network, at any time is with the best connection values, leading to fewer burdens on the incoming new neurons. The network error reduces as the training phase continues, in such cases the weights of the network are not stored; it is only when the error oscillates or moves through a local minimum, the connection values are stored.

5.2.1 Tracking the network weights

Consider a newly configured or partially trained network

The network weights and the error produced by this network are stored.

IF local minima are reached or the error starts to increase

Errors at these points are compared with the stored error value

IF the stored error value is more than the network error at the local minimum

Previously stored weights are replaced by the connection values corresponding to the local minima (Connection value minus latest change in the connection value) and the corresponding error is also stored.

IF the termination of the training phase is met

The network error at this point is compared to the stored error value, and the weights are replaced if it is less than the stored error value

The network weights are replaced by the stored values. If the network error has not reached the error tolerance, the network is partially trained.

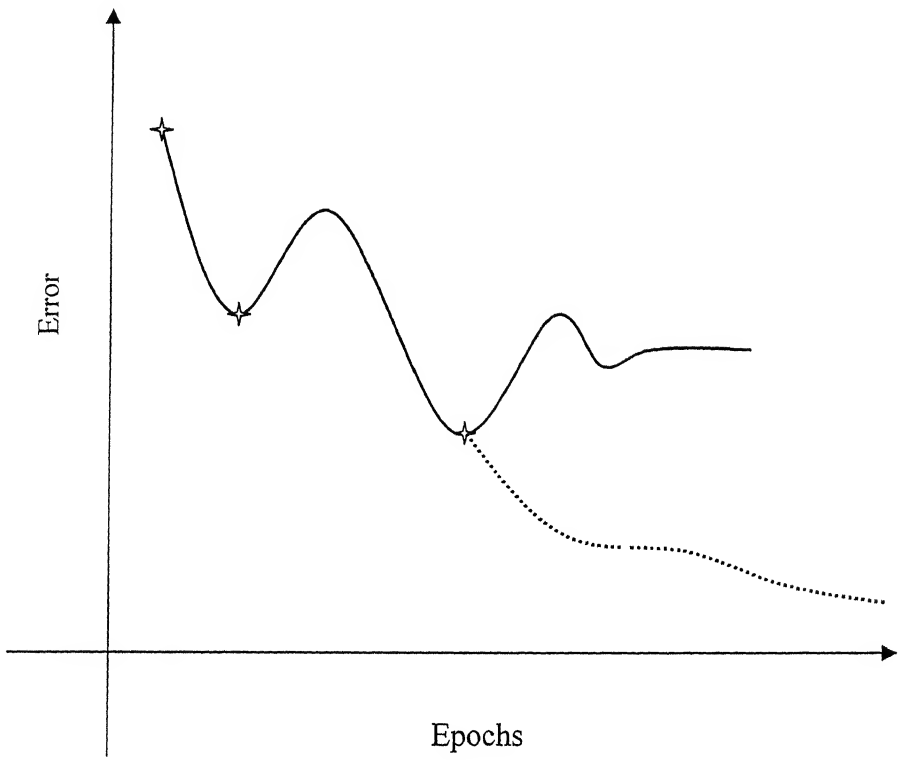


Fig 5.1 Tracking the network weights. ✧ Indicates the points at which network weights are stored Solid curve indicates the error produced by the existing network Dashed curve indicates the error produced by the network when a new neuron is cascaded

5.3 Extensions to Back propagation algorithms

There are number of algorithms which improve on back propagation's convergence properties. Momentum method is used to accelerate the convergence of the error back propagation algorithm. Quickprop algorithm, which employs a dynamic momentum rate, is one of the more successful algorithms for handling the step-size problem in back-propagation systems. RPROP algorithm, which performs a local adaptation of the weight-

updates according to the best behavior of the error function, is one of the faster converging BP variants

5.3.1 Quickprop

Quickprop computes the $\partial E/\partial w$ values just as in standard backprop, but instead of simple gradient descent, Quickprop uses a second-order method, related to Newton's method, to update the weights. Quickprop's weight-update procedure depends on two approximations: first, that small changes in one weight have relatively little effect on the error gradient observed at other weights; second, that the error function with respect to each weight is locally quadratic. For each weight, quickprop keeps a copy of $\partial E/\partial w(t-1)$ the slope computed during the previous training cycle, as well as $\partial E/\partial w(t)$, the current slope. It also retains $\Delta w(t-1)$, the change it made in this weight on the last update cycle. For each weight, independently, the two slopes and the step between them are used to define a parabola; then jump to the minimum point of this curve. Because of the approximations, this new point will probably not be precisely the minimum. As a single step in an iterative process, however, this algorithm seems to work very well. In practice, some complications must be added to the algorithm. A maximum growth factor μ is used to limit the increase of the step-size. The Quickprop algorithm is as given in Appendix A.

5.3.2 RPROP

RPROP modifies the size of the weight step taken adaptively; the mechanism for adaptation does not take into account the magnitude of the gradient $\partial E/\partial w$ as seen by a

particular weight, but only the sign of the gradient. This allows the step size to be adapted without having the size of the gradient interfere with the adaptation process.

The RPROP algorithm works by modifying each weight by an amount $\Delta_{ij}(t)$ termed the update value, in such a way to decrease the overall error. All updates values are initialized to the value Δ_0. The update value for a weight is modified in the following manner: if the gradient $(\partial E/\partial w(t))$ direction has remained the same in successive epochs, then the update value is multiplied by a value η^+ (which is greater than one). Similarly, if the gradient direction has changed, the update value is multiplied by a value η^- (which is less than one). This results in the update value for each weight adaptively growing or shrinking as a result of the sign of the gradient seen by that weight. There are two limits placed on the update value: a maximum Δ_{max}, and a minimum Δ_{min}. The initial update value Δ_0 is the single parameter, which requires setting by the user, with all algorithm parameters treated as constant. The RPROP algorithm is given is Appendix B.

5.3.3 SAPROP

While RPROP can be extremely fast in converging to a solution, it can often converge to poor local minima. SARPROP [6] attempts to address this problem by using the method of Simulated Annealing (SA). SA in general, involves the addition of noise to the parameters undergoing optimization. The amount of noise added is associated with a SA term, which decreases the effect of the noise as training progresses. The addition of noise allows the network to move in a direction, which is not necessarily the direction of

steepest descent. The benefit this provides is that it can help the network escape from local minima.

In SARPROP, noise is added to the standard RPROP weight update value when both the error gradient changes sign in successive epochs, and the magnitude of the update value is less than a value proportional to the SA term. The amount of noise added is proportional to the SA term. The reason for adding noise to the update value only when both the error gradient changes sign, and the update value is below a given setting, is to minimize the disturbance to the normal adaptation of the update value. Following this scheme means that the update value is only modified by noise when it has a relatively small value (indicating a number of previous gradient crossings). This can allow the weight to jump out of local minima, while minimizing the disturbance to the adaptation process.

SARPROP uses SA not only on the noise added to the weight updates, but also on the amount of weight decay the network uses. Weight decay is implemented in SARPROP by adding a penalty term to the error function, which results in a modification of the error gradient. The weight decay term in SARPROP is designed such that small valued weights decay more rapidly than larger weights. This decay term allows important functionality already learned by the network (and represented by the large weights) to be retained more easily.

The SA term applied to the weight decay in SARPROP results in the influence of the weight decay decreasing as training proceeds. The SARPROP error gradient [6] is shown below

$$\partial E/\partial w_{ij}^{SARPROP} = \partial E/\partial w_{ij} - 0.01 * w_{ij}/\left(1+w_{ij}^2\right) * SA$$

where $SA = 2^{-I*epoch}$ and T = temperature

The effect of this form of weight decay is to modify the error surface so that initially smaller weights are favored. As training progresses the weight decay magnitude is reduced, which modifies the error surface to allow for the easier growth of large weights. The modification of the error surface allows the network to explore regions of the error surface which were previously unavailable. The use of weight decay is a form of regularization, which has been found to improve generalization.

The SARPROP algorithm is as follows:

$\forall i, j \,.\, \Delta_{ij}(t) = \Delta_0$

$\forall i, j : \dfrac{\partial E}{\partial w_{ij}(t-1)} = 0$

REPEAT

Compute SARPROP Gradient $\dfrac{\partial E}{\partial w_{ij}(t)}$

For all weights and biases

IF $\dfrac{\partial E}{\partial w_{ij}(t-1)} * \dfrac{\partial E}{\partial w_{ij}(t)} > 0$

$\Delta_{ij}(t) = \textbf{Minimum}\left(\Delta_{ij}(t-1) * \eta^{+}, \Delta_{\max}\right)$

$\Delta w_{ij}(t) = \textbf{-sign}\left(\dfrac{\partial E}{\partial w_{ij}(t)}\right) * \Delta_{ij}(t)$

$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$

$\dfrac{\partial E}{\partial w_{ij}(t-1)} = \dfrac{\partial E}{\partial w_{ij}(t)}$

ELSEIF $\dfrac{\partial E}{\partial w_{ij}(t-1)} * \dfrac{\partial E}{\partial w_{ij}(t)} < 0$

IF $\left(\Delta_{ij}(t-1) < 0.4 * SA^2\right)$

$\Delta_{ij}(t) = \Delta_{ij}(t-1) * \eta^{-} + 0{,}8 * r * SA^2$

ELSE

$$\Delta_{ij}(t) = \Delta_{ij}(t-1) * \eta^{-}$$

$$\Delta_{ij}(t) = \textbf{Maximum}\left(\Delta_{ij}, \Delta_{\min}\right)$$

$$\frac{\partial E}{\partial w_{ij}(t-1)} = 0$$

ELSEIF $\dfrac{\partial E}{\partial w_{ij}(t-1)} * \dfrac{\partial E}{\partial w_{ij}(t)} = 0$

$$\Delta w_{ij}(t) = \textbf{-sign}\left(\frac{\partial E}{\partial w_{ij}(t)}\right) * \Delta_{ij}(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

$$\frac{\partial E}{\partial w_{ij}(t-1)} = \frac{\partial E}{\partial w_{ij}(t)}$$

UNTIL CONVERGED

In the SARPROP algorithm, r is a random number between zero and one. The only parameter requiring setting prior to training is the temperature T, a part of the SA term, which affects the speed by which the noise and weight decay are reduced. The optimal setting of this parameter is problem dependent. In general it was found the more complex the problem, the lower the temperature value required (and hence the slower the annealing process). With these enhancements to RPROP, the likelihood of escaping from the local minima is increased; there is still no guarantee. The RPROP/SARPROP algorithms sometimes result in oscillations in the error values. These situations are shown in Figures 5.2a and 5.3b. The following section addresses this situation.

5.3.4 Modifications to SAPROP (ReSARPROP)

Oscillations occur in the error values because of termination of SA mode. After considerable iterations (parameter dependent), the SA term that is added to the gradient term has negligible effect.

The modifications done to the algorithm are as follows When the error value oscillates, the current weight values are used as the new initial weights, so that any prior training knowledge is maintained. The SA terms are reset, which may allow the network to jump out from its current local minimum and perhaps converge to a better solution. This algorithm will be termed ReSARPROP [6].

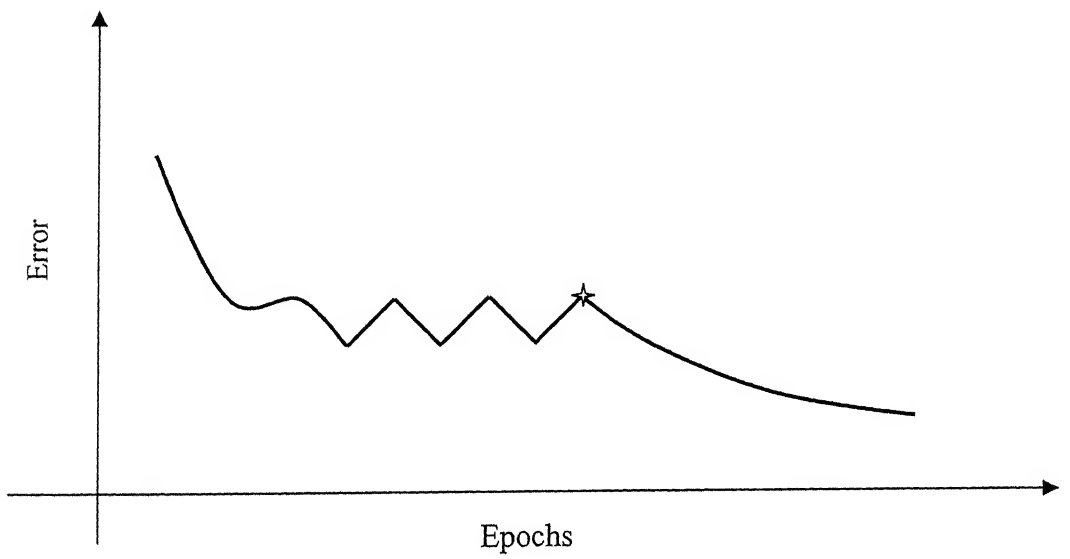


Fig 5 2a SAPROP Oscillations Case 1

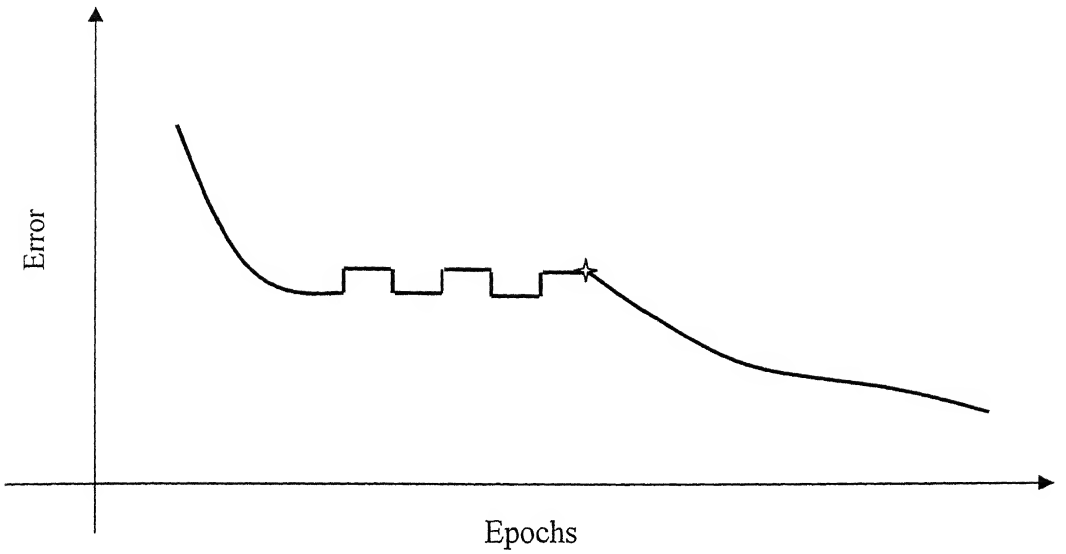


Fig 5.2b SAPROP Oscillations Case 2

The ReSARPROP algorithm is as follows:

Training is started by SARPROP

Keep track of error values. Store the latest six error values.

Check for the following conditions

```
IF (E [6]> E [5])
    IF (E [5]< E [4])
        IF (E [4]> E [3])
            IF (E [3]< E [2])
                IF (E [2]> E [1])
OR

  IF (E [6]= = E [5])
      IF (E [5]< E [4])
          IF (E [4]= = E [3])
              IF (E [3]> E [2])
                  IF (E [2]= = E [1])


THEN
        Reset the SA parameters.
```

Continue the training process.

SARPROP requires the Temperature parameter 'T' to be set prior to training, and the optimal value depends on the problem. Using ReSARPROP, the temperature can be initially set to give fast annealing. If a good solution has not been reached, this temperature can be reset to allow for slower annealing when the network is reset. The removal of the temperature parameter in ReSARPROP results in ReSARPROP being (user) parameter free. The training is started by SARPROP with the default value of 'T', and as ReSARPROP is applied, the value of 'T' is reset based on an exponential decrease.

Considering all these improvements, the algorithms developed are validated on several benchmark problems. These are discussed in the following chapter.

CHAPTER 6

RESULTS AND DISCUSSION

The algorithms developed have been tested on various benchmark problems and function mapping problems. The performance of algorithms is discussed

6.1 Exclusive-OR (XOR) by Cascor

Illustration of Cascor algorithm is presented here. XOR problem, which is linearly non-separable, is trained by Cascor with only the output layer as the initial architecture. Simple Back-Propagation (Pattern Mode) with momentum is used, where learning rate is 0.4 and momentum factor is 0.8. The error plot is shown in Fig 6.1.

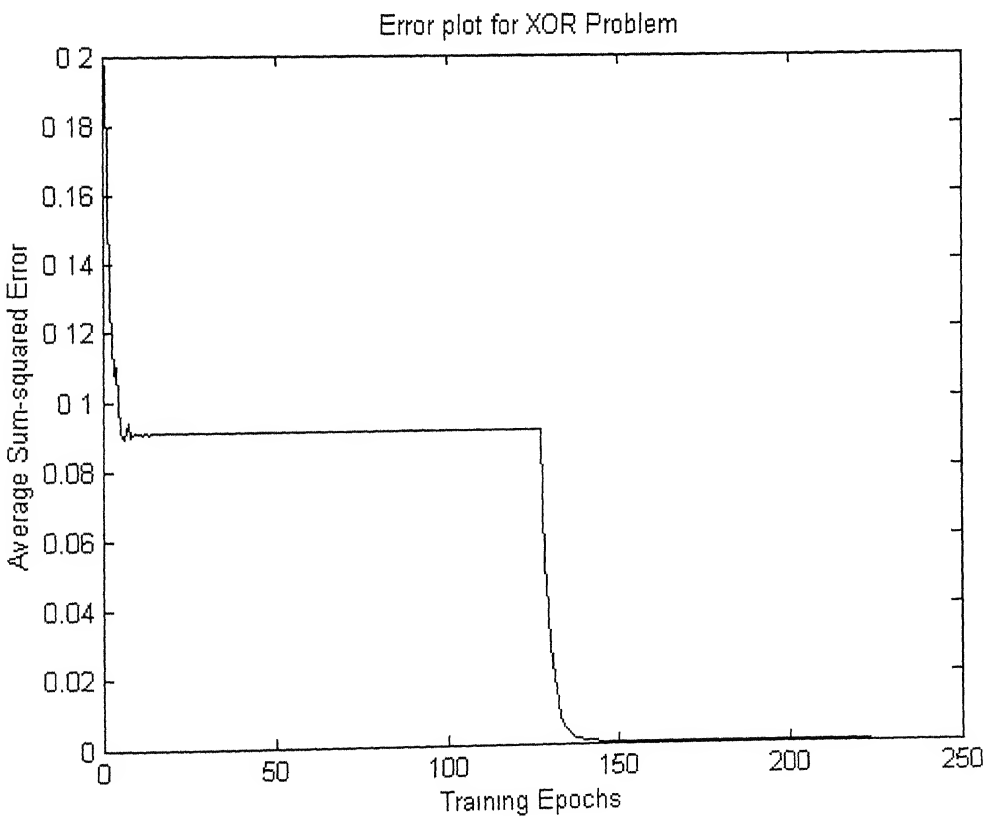


Fig 6.1 XOR by Cascor (BP-Pattern mode)

The error value is saturated at the iteration number 127, thereby a new neuron is considered. A pool of 10 neurons, all of them receiving the same inputs, but initialized in different sub-regions of weight space (-0.1 to 0.1, -0.2 to 0.2, ...-1 to +1) are created. They are trained to maximize the correlation between their output and the networks residual error (0.08 in this case). It has been observed that the neuron whose weights initialized in the sub-region -0.6 to 0.6 is emerged as a winner with the correlation value of 0.96 It is brought into the active network and the network is further trained. By the end of iteration number 224, the average sum-square value has reached the specified error tolerance value(1.0E-6).

Table 6.1 provides brief overview of the results for XOR problem by Cascor algorithm where different learning methods are applied. Learning rate, momentum factors are 0.4 and 0.8 for BP (Pattern, Batch) respectively. For Quickprop learning rate is 0.4, while for RPROP initial learning rate is 0.001.

Learning Methodology	Hidden units	Approximate iterations	Iterations for Maximizing correlation
BP – Pattern mode	1	127+97	Average number of iterations per candidate unit: 1000
BP – Batch mode	1	150+100	
Quickprop	1	55+110	
RPROP	1	45+40	

Table 6.1 XOR by Cascor

6.2 Three-Parity by Cascor

3-Parity problem is presented to Cascor; the results are shown in Table 6.2. The specified error tolerance is 1.0E-6. The corresponding error plots are shown in Fig 6.2 1 and Fig 6.2.2. Learning rate, momentum factors are 0 4 and 0.8 for BP(Pattern, Batch) respectively. For Quickprop learning rate is 0.4; while for RPROP and SAPROP, initial learning rate is 0 001.

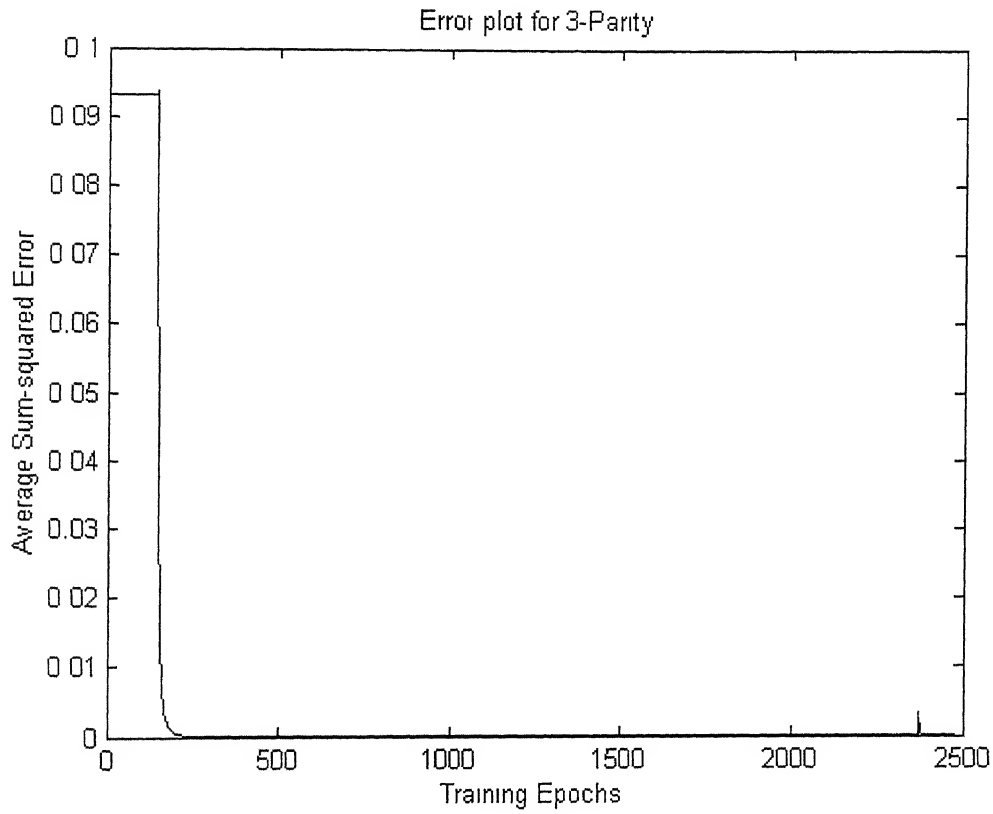


Fig 6.2.1 3-Parity by Cascor

Learning Methodology	Hidden units	Approximate ıterations	Error Plot
BP – Pattern mode	2	300+2030+200	Fig 6 2
BP – Batch mode	1	180+450	Fig 6.3A
Quickprop	1	180+290	Fig 6.3B
RPROP	1	120+660	Fig 6.3C
SARPROP	1	130+720	Fig 6.3D

Table 6.2 Three-Parity by Cascor

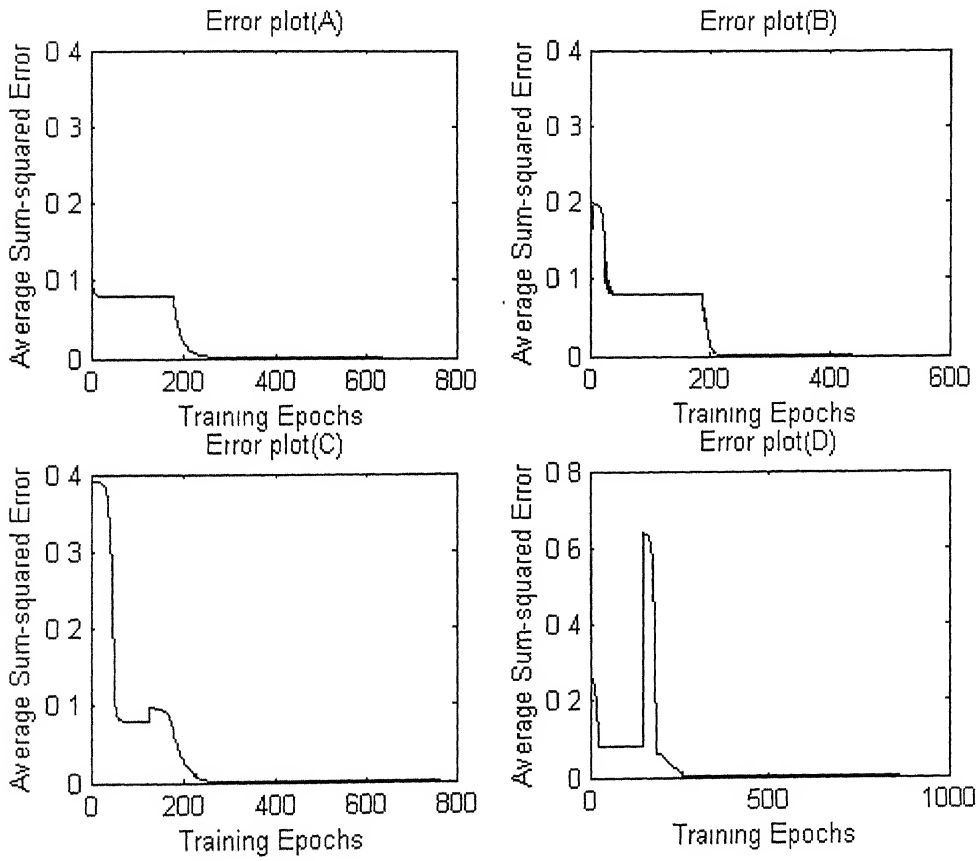


Fig 6.2.2 A,B,C,D 3-Parity by Cascor

6.3 Sin(x)*Sin(y) Problem by CasCor

Sin(x)*Sin(y) is a function approximation problem. It has been observed that for solving this problem, the values of learning rate and momentum factor should be small. The values taken are: 0.001 as the learning rate and 0.005 as the momentum factor. In case of RPROP and SARPROP, the initial learning rate is 0.0001. For the specified error tolerance of 1.0E-5, the results are tabulated in Table 6.3.

Learning Methodology	Hidden units	Iterations Per unit (approximate)
BP – Batch mode	34	43000
Quickprop	28	35000
RPROP	28	30000
SARPROP	28	30000

Table 6.3 Sin(x)*Sin(y) by Cascor

More the number of patterns, more the time it takes to maximize the correlation value. The number of candidate units taken in a pool is 10. The plots of sin(x)*sin(y)-BP Batch mode are shown in figures 6.3.1, 6.3.2 and 6.3.3. It has been observed that the last 10 to 12 hidden units are for fine-tuning of the error. The addition of hidden units to the network where already 15 to 18 units are cascaded will reduce the error at a slower pace. The experiment has been performed by reloading the trained network and by increasing the error tolerance value and the maximum number of hidden units that can be added.

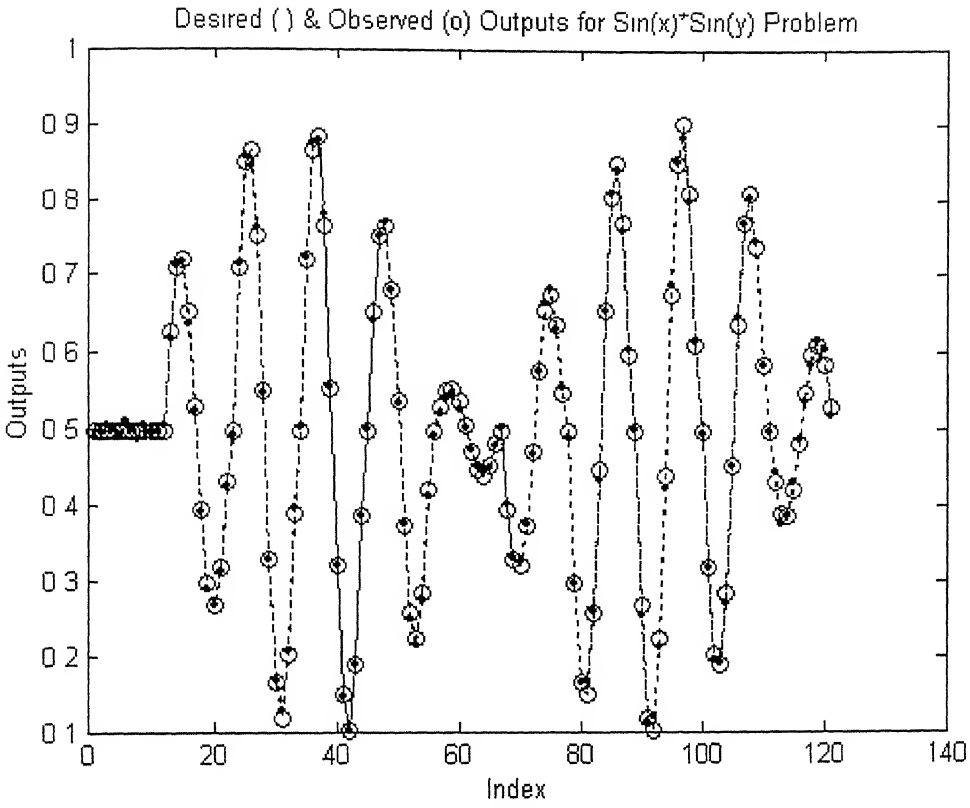


Fig 6.3.1 Desired Vs Observed outputs- Sin(x)*Sin(y)-Cascor

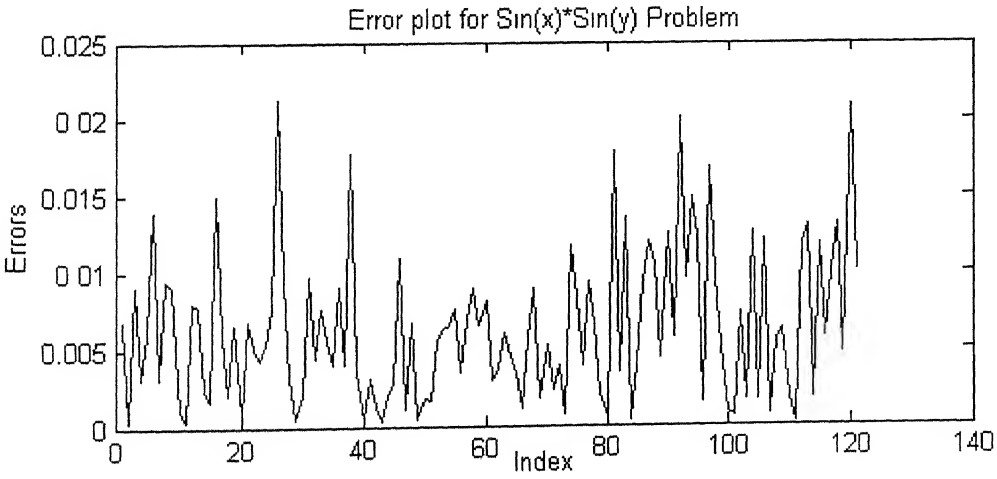


Fig 6.3.2 Errors at each pattern--Sin(x)*Sin(y)-Cascor

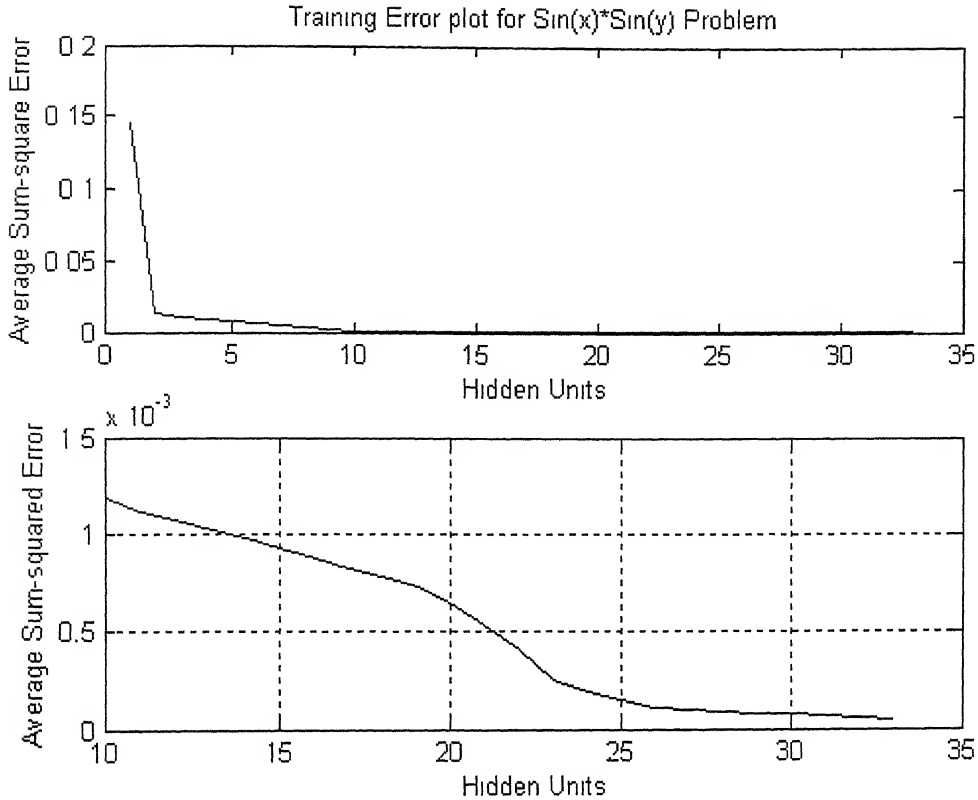


Fig 6.3.3 Error Vs Hidden units -Sin(x)*Sin(y)-Cascor

6.4 Two-Spiral Problem by CasCor

For 2-Spiral problem, the inputs and outputs are normalized in different ranges. The outputs are normalized in the range –0.5 and +0.5(the first spiral outputs are classified as –0.5, where as the second spiral outputs are classified as +0.5). The activation function used is bipolar sigmoid. The inputs are normalized in the range 0 1 and 0.9. The results are tabulated in Table 6.4 and the plots by BP-Batch mode are shown in figures 6.4.

Learning Methodology	Hidden units	Iterations Per unit (approximate)
BP – Batch mode	24	50000
Quickprop	23	40000
RPROP	21	40000
SARPROP	21	40000

Table 6.4 Two-Spiral problem by Cascor

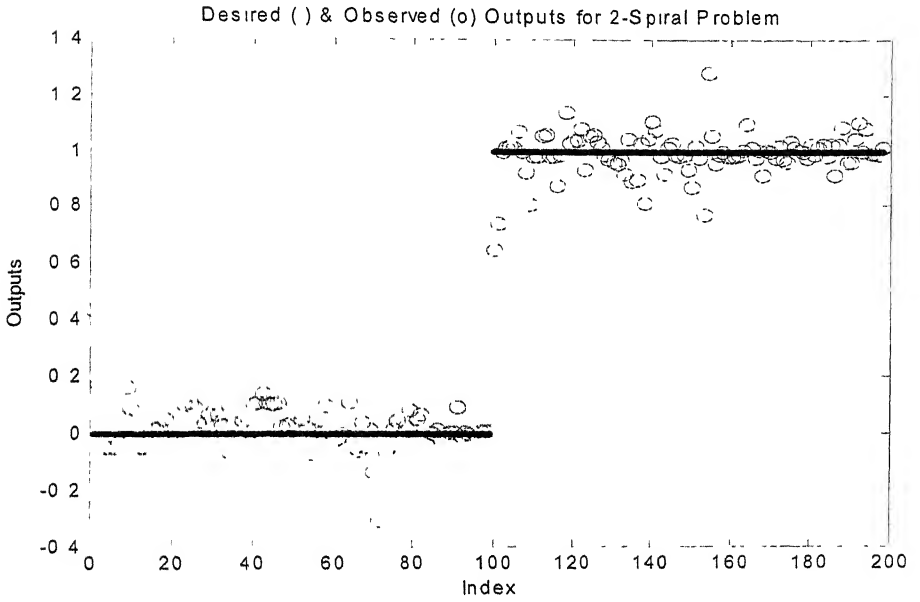


Fig 6.4.1 Desired Vs Observed outputs-Two-Spiral Problem-Cascor

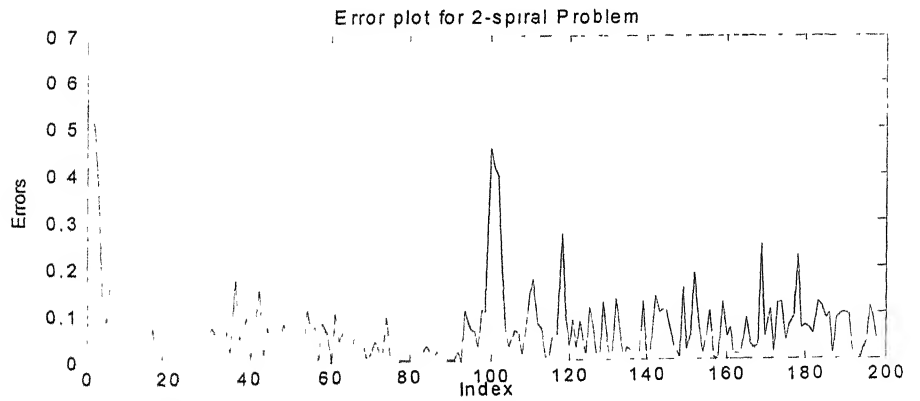


Fig 6.4.2 Errors at each pattern-Two-Spiral Problem-Cascor

6.5 Predicting Chaotic Dynamics by Cascor

The time series used in this simulation is generated by the chaotic Mackey-Glass differential delay equation [7] defined below,

$$\dot{x}(t) = \frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)} - 0.1x(t). \tag{6.1}$$

The prediction of future values of this time series is a benchmark problem, which has been considered by a number of researchers. The goal of the task is to use known values up to the point x=t to predict the value at some point in the future x=t+P. The standard method for this type of prediction is to create a mapping from D points of the time series spaced *d* apart, that is, (x(t-(D-1)*d*),...,x(t-*d*),x(t)), to a predicted future value x(t+P). The value D=3, D=4 and D=6 has been used in this simulation.

From the Mackey-Glass time series x(t), 1000 input-output data pairs have been extracted, of which the first 500 have been used for training and the rest for testing. The simulation results are shown in Table 6.5. Figures 6.5.1 and 6.5 3 are the plots for the test data when D=3 and D=4 respectively. Figures 6.5.2 and 6.5.4 are the corresponding plots of errors at each pattern, where all the 1000 patterns are presented.

	Training Methodology	Hidden Units		Training Methodology	Hidden Units		Training Methodology	Hidden Units
D=3	BP - Batch	9	D=4	BP - Batch	6	D=6	BP - Batch	4
	Quickprop	8		Quickprop	6		Quickprop	4
	RPROP	8		RPROP	6		RPROP	4
	SARPROP	8		SARPROP	6		SARPROP	4

Table 6.5 Mackey-Glass time series prediction by Cascor

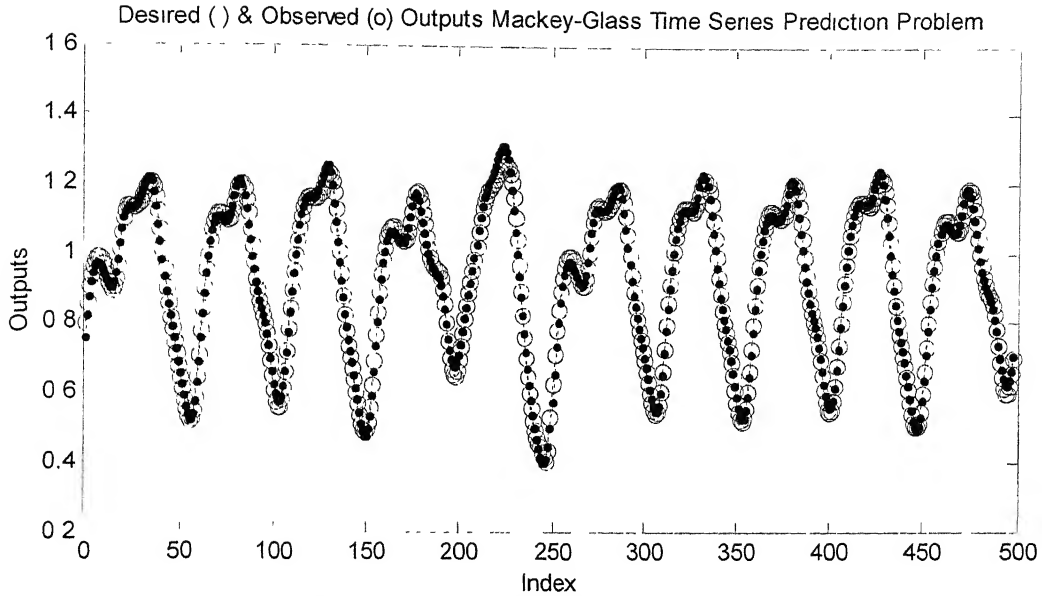


Fig 6.5.1 Mc-Glass Time Prediction (D=3)-Test Results-Cascor

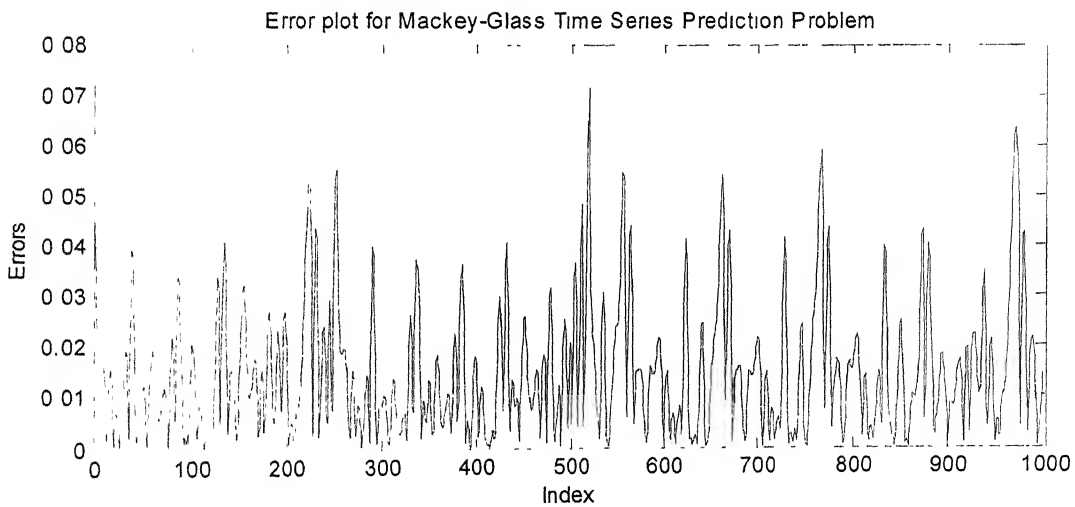


Fig 6.5.2 Mc-Glass Time Prediction (D=3)-all patterns-cascor

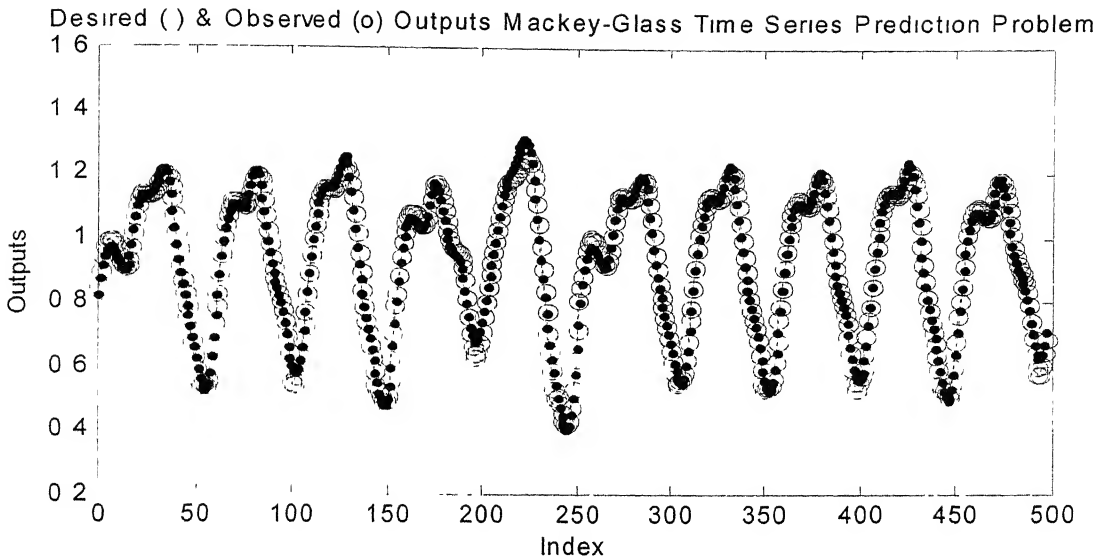


Fig 6.5.3 Mc-Glass Time Prediction (D=4)-Test Results-Cascor

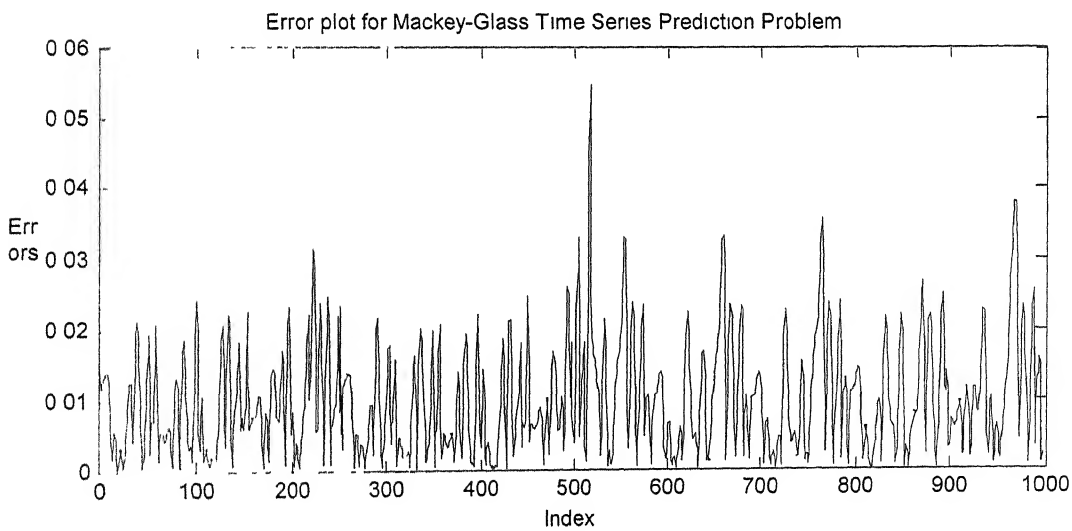


Fig 6.5.2 Mc-Glass Time Prediction (D=4)-all patterns-cascor

6.6 Results by Caserr and Casall

The results of caserr on XOR, 3-parity and 4-parity are shown in Table 6.6.1. Average number of iterations per candidate unit before it comes into the active network is 1000.

	Learning Methodology	Hidden units	Approximate iterations
XOR	BP – Pattern mode	1	127+(1 to 20)
	BP – Batch mode	1	150+(1 to 20)
	Quickprop	1	55+(1 to 20)
	RPROP	1	45+(1 to 20)
3-Parity	BP – Pattern mode	2	300+1000+(1 to200)
	BP – Batch mode	1	180+(100 to 600)
	Quickprop	1	180+(100 to 600)
	RPROP	1	120+(100 to 600)
	SARPROP	1	130+(100 to 600)
4-Parity	BP – Pattern mode	4	300+(1000)x3+(1 to500)
	BP – Batch mode	4	180+(1000)x3+(1 to500)
	Quickprop	4	180+(1000)x3+(1 to500)
	RPROP	4	120+(1000)x3+(1 to500)
	SARPROP	4	130+(1000)x3+(1 to500)

Table 6.6.1 Caserr results

The performance of Casall algorithm can be differentiated when there is requirement of more than two hidden units.

The results of 4-parity are shown in Table 6.2. The results are taken by varying the upper limit of the number of neurons permitted in each hidden layer.

	Architecture	Training Methodology
4-Parity	[6]	BP - Batch
	[3 2]	Quickprop
	[4 2]	BP -Batch
	[2 2 2 2]	BP - Batch

Table 6.6.2 Four-Parity by Casall

Problem	Algorithm	Architecture {Hidden)	Training method	Iterations Per unit (approximate)
Sin(x)*Sin(y)	Caserr	24 units	SARPROP	20000
		26 units	Quickprop	24000
		27 units	BP-Batch	30000
	Casall	[15 12]	BP-Batch	33000
		[10 10 4]	BP-Batch	28000
		[6 6 6 6 2]	Quickprop	34000
		[20 8]	BP-Batch	28000

Table 6.6.3 Sin(x)*Sin(y) by Caserr and Casall

The need of additional neurons arises when the present architecture is not capable of solving the problem. The improved variants of Back Propagation: Quickprop and RPROP work effectively if the architecture is capable of solving the problem. It has been observed that the number of neurons added to the network is more or less same for a particular problem, irrespective of the training method

In case of CasAll algorithm, the results are taken by varying the maximum number of units permitted in each layer. Various architectures are resulted when the upper limit on the maximum neurons is altered.

Problem	Algorithm	Architecture (Hidden)	Training method	Iterations Per unit (approximate)
Two-Spiral Problem	Casall	[10 8]	Quickprop	25000
		[20 6]	BP-Batch	34000
		[15 12]	BP-Batch	35000
	Caserr	22	BP-Batch	30000
		21	Quickprop	26000
		18	RPROP	20000
		18	SARPROP	20000

Table 6.6.4 Two-Spiral by Caserr and Casall

Problem	Training Method	Iterations
XOR	BP-Batch	1120
	SARPROP	78
3-Parity	BP-Batch	1400
	SARPROP	580

Table 6 7.1 GN: XOR and 3-Parity

6.7.2 Four-Parity

Tables 6.7.2, 6.7.2a and 6.7.2b give the architecture used and the results for 4-Parity problem. The plots are shown in figures 6.7.2a and 6 7.2b.

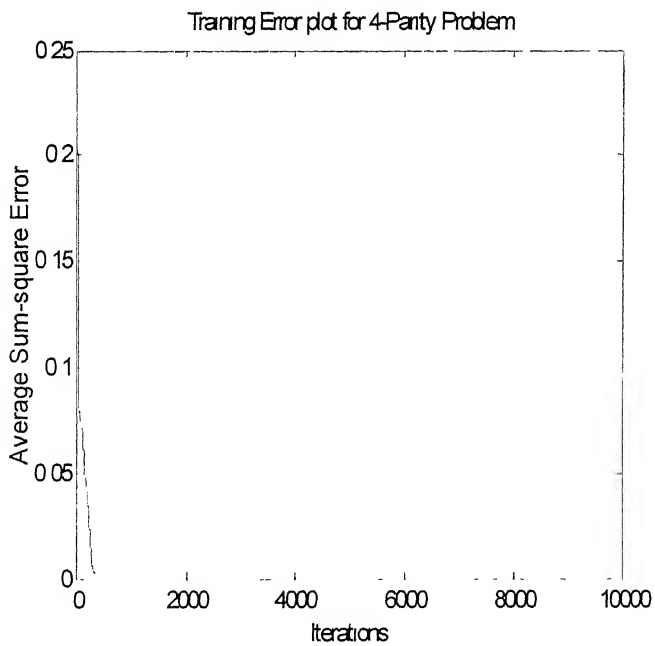


Fig 6.7.2a

Desired	Observed
0	6 99E-04
1	0 989255
1	0 999493
0	0 002212
1	0 996723
0	0 002201
0	-0 00188
1	0.99833
1	1 0125
0	-8.01E-05
0	-5 89E-04
1	1 000161
0	1 22E-04
1	0 999514
1	1 0158
0	-0 00133

Table 6.7.2a

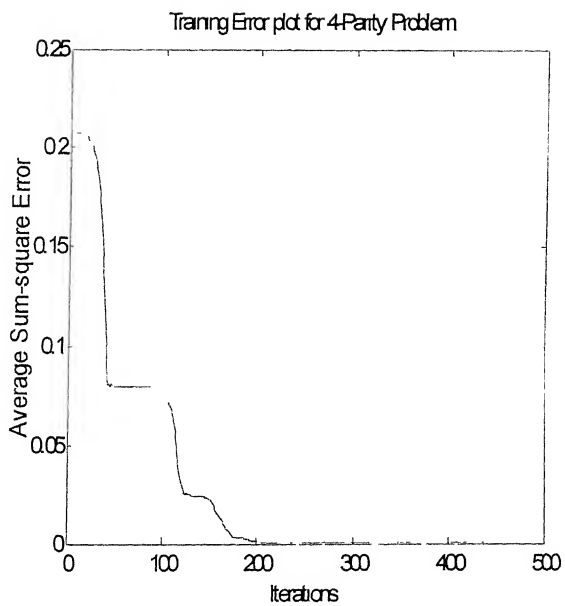


Fig 6.7.2b

Desired	Observed
0	-7 53E-05
1	1 000478
1	1 00011
0	-8.96E-04
1	1 002046
0	-4 46E-04
0	0 001449
1	0 99697
1	0 999244
0	0 001079
0	-0 00104
1	1 003234
0	-7 85E-04
1	0 996263
1	0 997769
0	4 00E-04

Table 6.7.2b

	Hidden Layer-1 with two neurons		Training Method	Iterations	Error
	Aggregations	Activations			
4-Parity	Summation, Sum of Products (quadratic)	Sigmoid	SARPROP	10000	3.20E-4
	Summation, Sum of Products (quadratic) and Radial Basis Term	Sigmoid Tan Hyperbolic	SARPROP	500	9.94E-7

Table 6.7.2 GN: Four-Parity

6.7.3 Modeling a three input nonlinear function

The training data for this problem are obtained from

$$output = \left(1 + x^{0.5} + y^{-1} + z^{-1.5}\right)^2 \tag{6.2}$$

216 training data and 125 checking data have been generated uniformly from the input ranges [1,6]*[1,6]*[1,6] and [1.5,5.5]*[1.5,5 5]*[1.5,5.5], respectively.

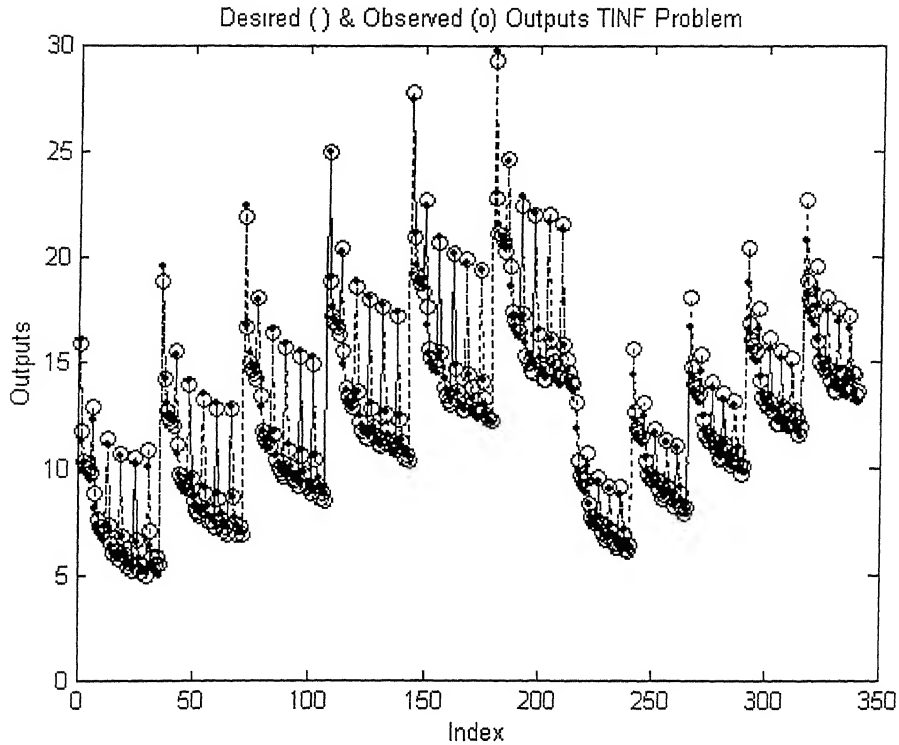


Fig 6.7.3a TINF outputs

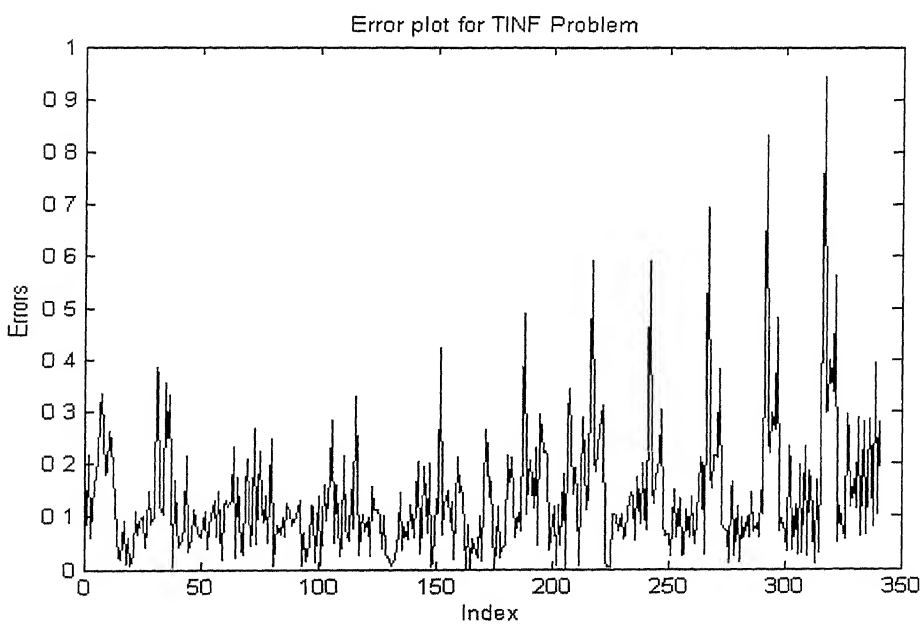


Fig 6.7.3 b TINF Error Plot

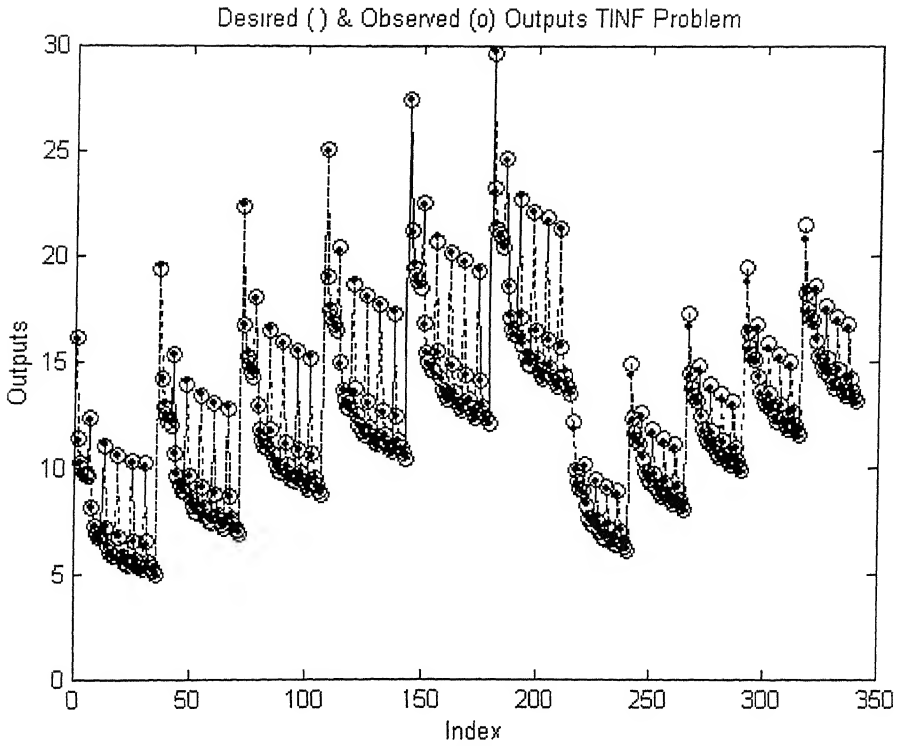


Fig 6.3.7c TINF Outputs

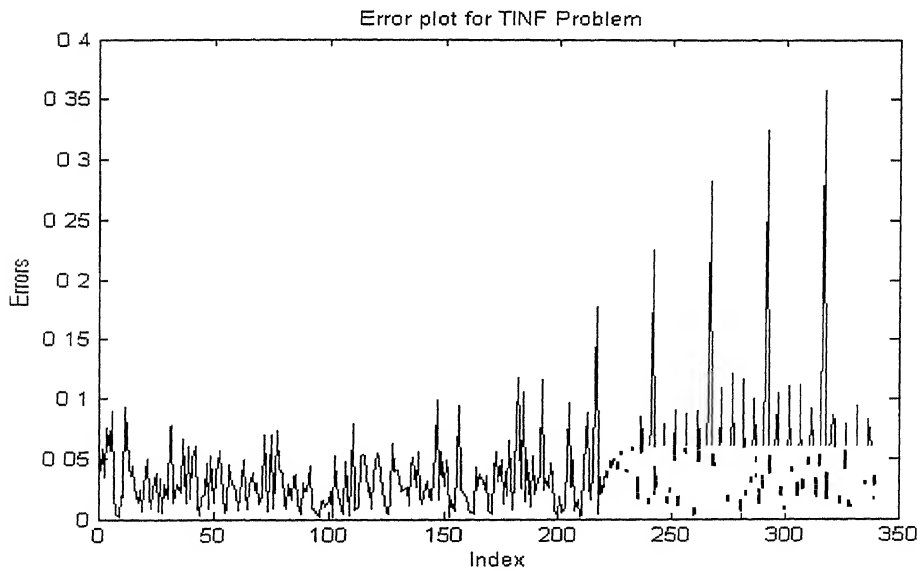


Fig 6.7.3d TINF Error Plot

The architecture used is [3 2 1]. In case-1, the hidden units have two aggregation functions (Summation, Sum of Product of two terms), while the output neuron has only

summation aggregation. The activation function is bipolar sigmoid. The results are shown in figures 6.7.3a and 6.7.3b. In case-2, all the neurons have three aggregation functions (Summation, Sum of Product of two terms and Radial basis term) and two activation functions (Sigmoidal and Tan Hyperbolic). The results are shown in figures 6.7.3c and 6.7.3d. The results of case-2 are better than that of case-1.

6.7.4 Sin(x)*Sin(y) results

The results are shown in Table 6.7.4. The plots of case 2 are shown in fig 6.7.4.

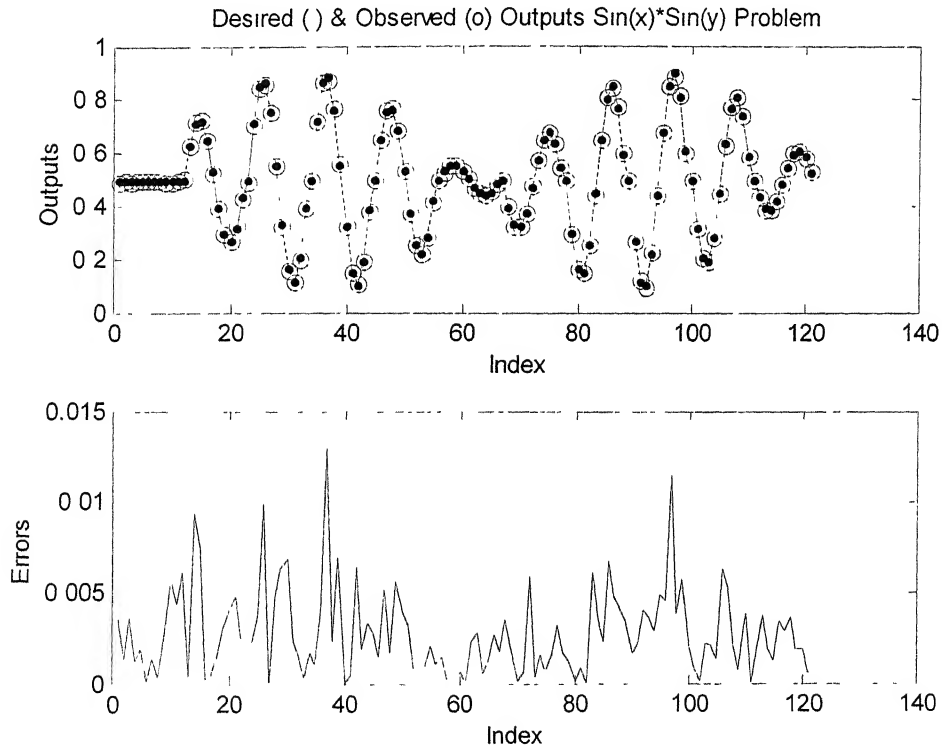


Fig 6.7.4 Sin(x)*Sin(y) plots

Case	Architecture (Hidden)	Aggregations	Activations	Learning Method
1	[6 4]	Summation, SOP2 And RB terms	Sigmoid, Tan Hyperbolic	BP-Batch
2	[10]	Summation, SOP2 And RB terms	Sigmoid, Tan Hyperbolic	SARPROP
3	[8 6]	Summation in the first layer, SOP2 in the second layer Summation in the output layer	Sigmoid	BP-Batch, SARPROP

Table 6.7.4 GN: Sin(x)*Sin(y)

6.7.5 Two-Spiral results

The results are shown in Table 6.7.5. The error plots of Case:1 and Case:3 (SARPROP) are shown in figures 6.7.5a and 6.7.5b.

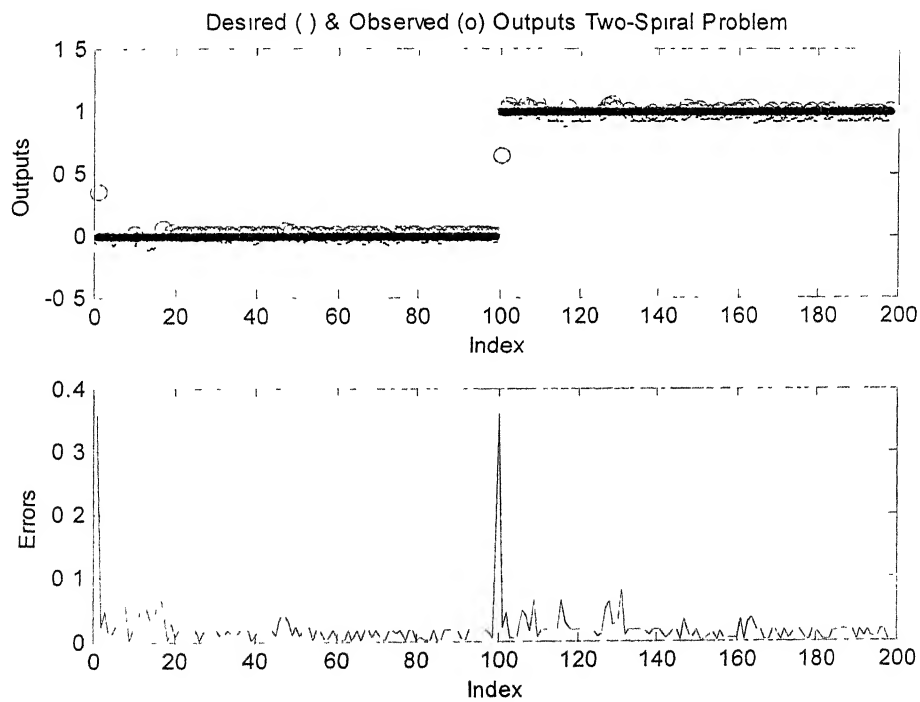


Fig 6.7.5a Two-Spiral plots

Case	Architecture (Hidden)	Aggregations	Activations	Learning Method
1	[5 5]	Summation, SOP2 And RB terms	Sigmoid, Tan Hyperbolic	BP-Batch
2	[8]	Summation, SOP2 And RB terms	Sigmoid, Tan Hyperbolic	SARPROP
3	[8 6]	Summation in the first layer, SOP2 in the second layer Summation in the output layer	Sigmoid	BP-Batch, SARPROP
4	[4 8]	Summation in the first layer, SOP2 in the second layer Summation in the output layer	Sigmoid	BP-Batch, SARPROP

Table 6.7.5 GN: Two-Spiral Problem

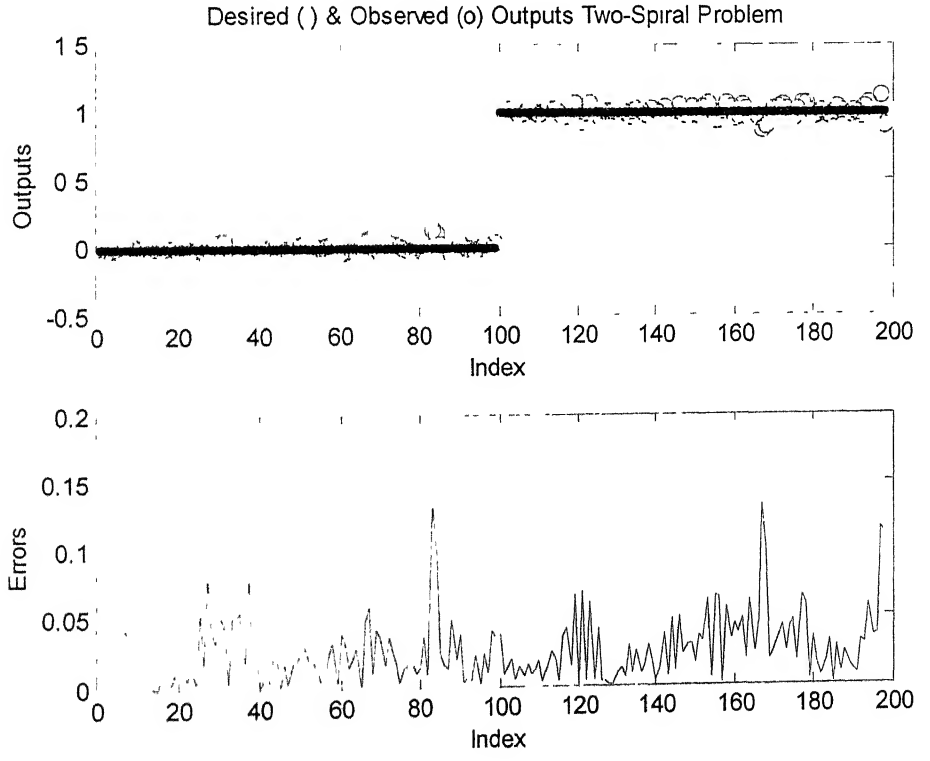


Fig 6.7.5b Two-Spiral plots

6.7.6 Predicting Chaotic Dynamics

Mc-Glass Time Prediction Series (with D=3, 4 and 6) are solved by a single generalized neuron. The Aggregations used are summation, SOP2 and RB terms. The activations used are sigmoid and tan hyperbolic functions. Figures 6.7.6a, 6.7.6c and 6.7.6e are the plots for the test data when D=3, D=4 and D=5 respectively. Figures 6 7.6b, 6.7.6d and 6.7.6f are the corresponding plots of errors at each pattern, where all the 1000 patterns are presented.

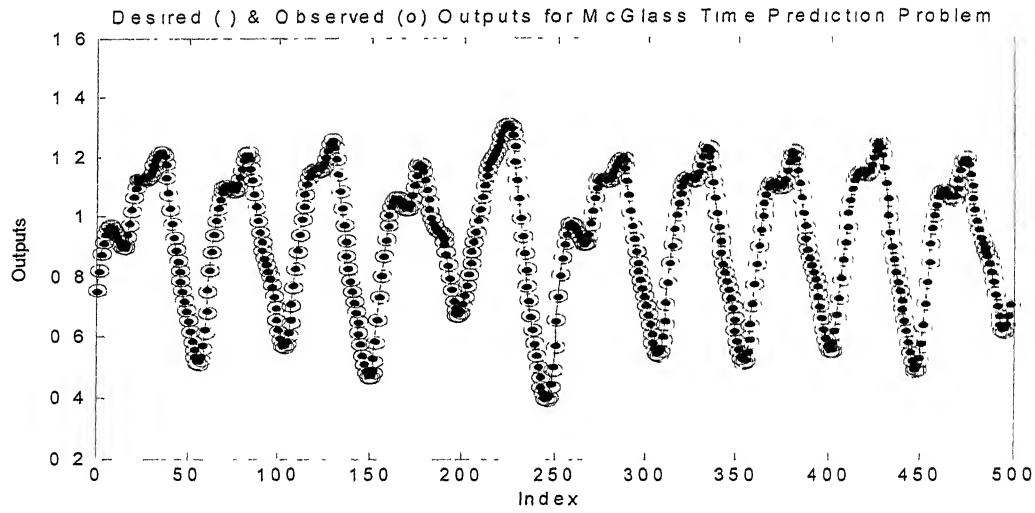


Fig 6.7.6a Mc-Glass Time Series Prediction (D=3)-Test Results

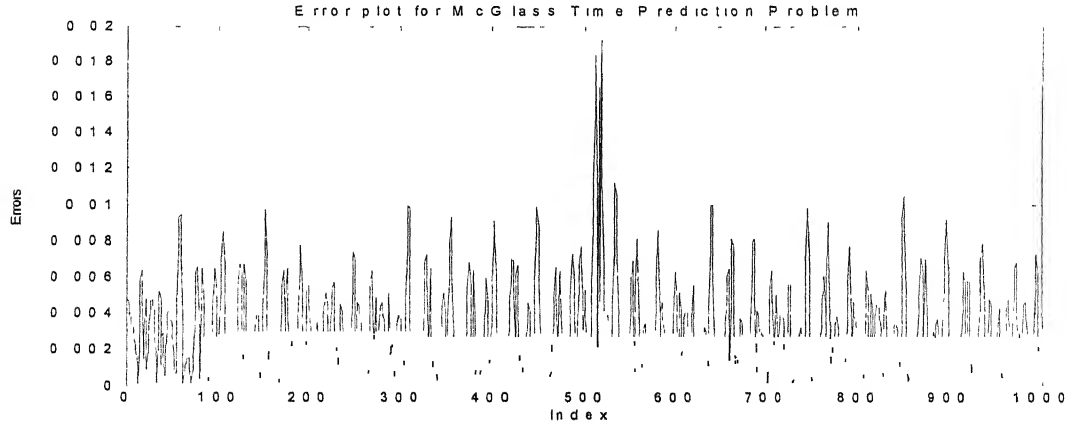


Fig 6.7.6 b Mc-Glass Time Series Prediction (D=3)-All Patterns-Errors

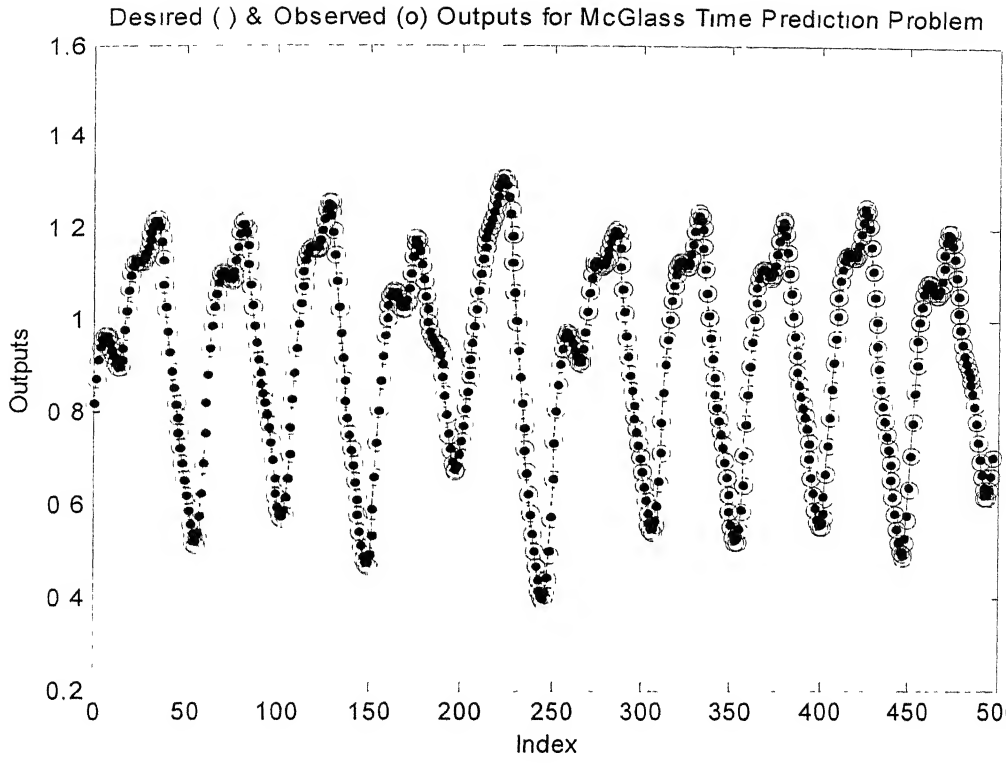


Fig 6.7.6c Mc-Glass Time Series Prediction (D=4)-Test results

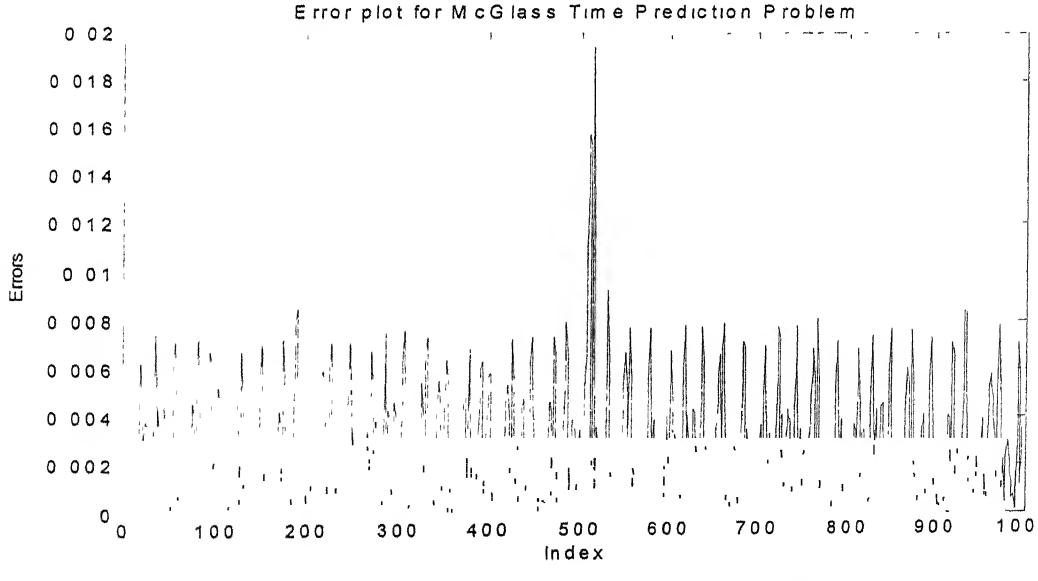


Fig 6.7.6d Mc-Glass Time Series Prediction (D=4)-All Patterns-Errors

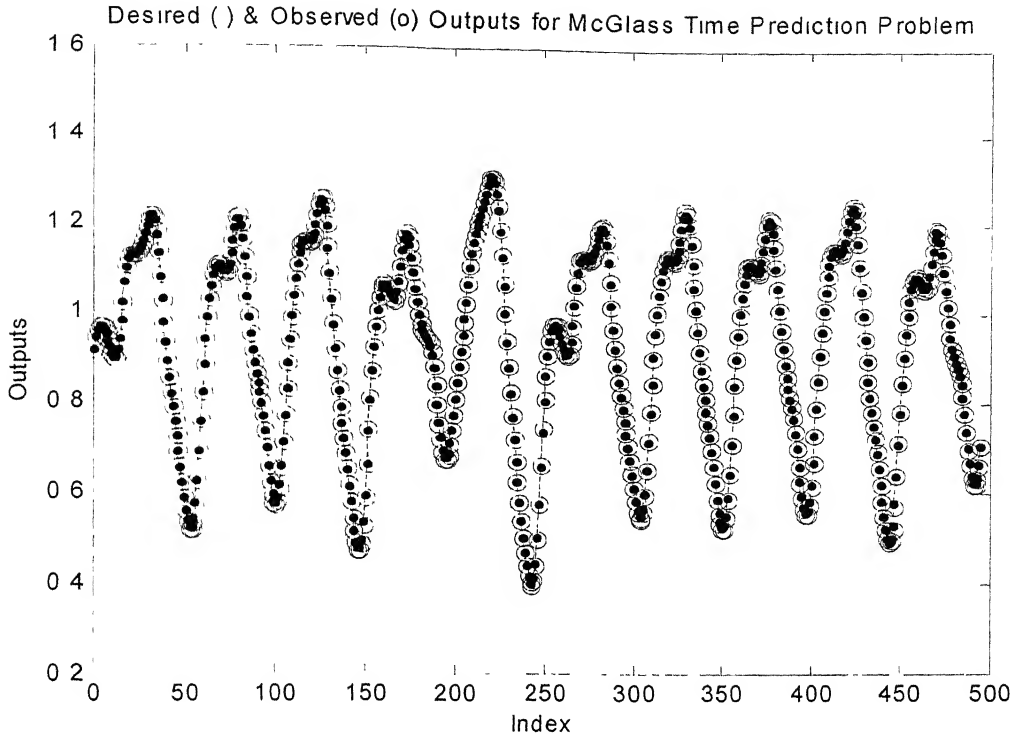


Fig 6.7.6e Mc-Glass Time Series Prediction (D=6)-Test Results

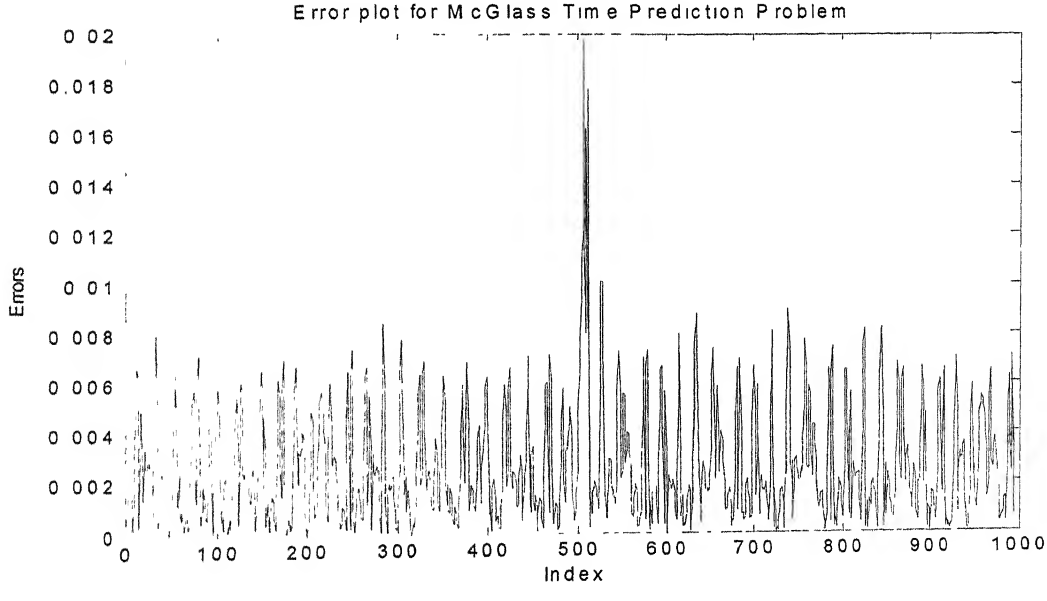


Fig 6.7.6f Mc-Glass Time Series Prediction (D=6)-All Patterns-Errors

6.8 Results by CasAny

The CasAny Algorithm, with the choice of three aggregation functions and two activations is tested. The three aggregation functions are Summation, SOP2 and Radial Basis terms. The two activation functions are Bi-polar sigmoid and Tan hyperbolic functions. For the three aggregations and two activations, there are 21 combinations. A pool of 21 candidate units' results, from which a unit that best suits in the network is picked up. All the weights (connection values) are initialized randomly. Therefore, different architectures are resulted on various runs. The results for XOR, 3-parity and 4-parity are shown in Table 6.8.

Problem	Architecture
XOR	One hidden unit. With RB term and sigmoidal activation function
	One hidden unit. With two aggregations· Summation and SOP2 and Sigmoidal Activation function.
	One hidden unit. Summation term, and two activation functions: Sigmoidal and Tan Hyperbolic functions.
3-Parity	One Hidden layer with two neurons First Neuron: Summation, RB term and tan hyperbolic. Second neuron: SOP2, Sigmoid and tan hyperbolic.
	One Hidden unit. All the three aggregations and sigmoidal activation function.
	Two Hidden layers with one unit each. RB term, tan hyperbolic in the first neuron; SOP2 and RB term, sigmoid transfer function.
4-Parity	Two hidden Layers. [3 2].{(Summation, SOP2,tanh),(Summation, sigmoidal), (RB, SOP2, sigmoidal), (summation, RB, tanh) and (SOP2, sigmoidal)}

Table 6.8 XOR, 3-Parity and 4-Parity Problems

As the number of aggregation functions and activation functions increases, the number of combinations increases. Selecting a unit from the pool where there are many neurons, increase the time complexity. It has been observed that in some cases, addition of new unit increase the error value. In such cases, the previously trained network is loaded and further trained by SAPROP, and then a new unit is added if required.

The simulation results of Sin(x)*Sin(y) is shown below. Output Layer is taken as layer '0'. First Neuron in a layer starts from number '0'. When the number of specified neurons is 10, the following architecture with 8 neurons is resulted.

```
Aggregation Functions in layer :0 Neuron . 0 are :1
1 : Summation

Activation Functions in layer :0 Neuron : 0 are :1
1 : Bipolar Sigmoid

 No. of Neurons in Layer: 1  : 8

Aggregation Functions in layer: 1 Neuron : 0 are :2
1 : SOP2
2 : Radial basis term

Activation Functions in layer: 1 Neuron : 0 are :1
1 : Tan hyperbolic

Aggregation Functions in layer: 1 Neuron : 1 are :1
1 : SOP2

Activation Functions in layer: 1 Neuron : 1 are :1
1 : Tan hyperbolic

Aggregation Functions in layer: 1 Neuron : 2 are :1
1 : SOP2

Activation Functions in layer: 1 Neuron : 2 are :1
1 : Bipolar Sigmoid

Aggregation Functions in layer. 1 Neuron : 3 are :2
1 : Summation
2 : SOP2

Activation Functions in layer: 1 Neuron : 3 are :2
1 : Tan hyperbolic
2 : Bipolar Sigmoid

Aggregation Functions in layer: 1 Neuron : 4 are :1
1 : Summation
```

```
Activation Functions in layer: 1 Neuron : 4 are :1
1 : Tan hyperbolic

Aggregation Functions in layer: 1 Neuron : 5 are :3
1 : Summation
2 : SOP2
3 : Radial basis term

Activation Functions in layer: 1 Neuron : 5 are :1
1 : Tan hyperbolic

Aggregation Functions in layer: 1 Neuron : 6 are :1
1 : Radial basis term

Activation Functions in layer: 1 Neuron : 6 are :2
1 : Tan hyperbolic
2 : Bipolar Sigmoid

Aggregation Functions in layer: 1 Neuron : 7 are :3
1 : Summation
2 : SOP2
3 : Radial basis term

Activation Functions in layer: 1 Neuron : 7 are :2
1 : Tan hyperbolic
2 : Bipolar Sigmoid
```

For the 2-Spiral Problem, the following architecture is resulted. Inputs and outputs are normalized in the ranges [-0.9,0.9] and [-0.5,0.5] respectively. Two hidden layers with the maximum of eight neurons in each are specified as the limits.

```
No. of Neurons in Layer: 0  : 1

Aggregation Functions in layer:0 Neuron: 0 are :1
1 : Summation

Activation Functions in layer:0 Neuron: 0 are :1
1 : Bipolar Sigmoid

 No. of Neurons in Layer: 1  : 8

Aggregation Functions in layer·1 Neuron: 0 are :2
1 : Summation
2 : SOP2

Activation Functions in layer:1 Neuron: 0 are :1
1 : Bipolar Sigmoid

Aggregation Functions in layer:1 Neuron: 1 are :1
1 : Summation
```

```
Activation Functions in layer:1 Neuron: 1 are :1
1 : Bipolar Sigmoid

Aggregation Functions in layer:1 Neuron: 2 are :2
1 : Summation
2 : SOP2

Activation Functions in layer:1 Neuron: 2 are :1
1 : Tan hyperbolic

Aggregation Functions in layer:1 Neuron: 3 are :2
1 : SOP2
2 : Radial basis term

Activation Functions in layer:1 Neuron: 3 are :2
1 : Bipolar Sigmoid
2 : Tan hyperbolic

Aggregation Functions in layer:1 Neuron: 4 are :1
1 : Summation

Activation Functions in layer:1 Neuron: 4 are :1
1 : Bipolar Sigmoid

Aggregation Functions in layer:1 Neuron: 5 are :3
1 : Summation
2 : SOP2
3 : Radial basis term

Activation Functions in layer:1 Neuron: 5 are :1
1 : Tan hyperbolic

Aggregation Functions in layer:1 Neuron: 6 are :1
1 : SOP2

Activation Functions in layer:1 Neuron: 6 are :1
1 : Tan hyperbolic

Aggregation Functions in layer:1 Neuron: 7 are :3
1 : Summation
2 : SOP2
3 : Radial basis term

Activation Functions in layer:1 Neuron: 7 are :1
1 : Tan hyperbolic


 No. of Neurons in Layer: 2  : 8

Aggregation Functions in layer:2 Neuron: 0 are :1
1 : Summation

Activation Functions in layer:2 Neuron: 0 are  1
1 : Bipolar Sigmoid

Aggregation Functions in layer:2 Neuron: 1 are :1
1 : Summation
```

```
Activation Functions in layer:2 Neuron: 1 are :1
1 : Bipolar Sigmoid

Aggregation Functions in layer:2 Neuron. 2 are .1
1 : Summation

Activation Functions in layer:2 Neuron: 2 are :1
1 : Tan hyperbolic

Aggregation Functions in layer:2 Neuron  3 are .3
1 : Summation
2 : SOP2
3 : Radial basis term

Activation Functions in layer:2 Neuron: 3 are :1
1 : Tan hyperbolic

Aggregation Functions in layer:2 Neuron  4 are ·2
1 : Summation
2 : SOP2


Activation Functions in layer:2 Neuron: 4 are :1
1 : Tan hyperbolic

Aggregation Functions in layer:2 Neuron: 5 are :2
1 : Summation
2 : SOP2

Activation Functions in layer:2 Neuron· 5 are :1
1 : Tan hyperbolic
```

The other benchmark problems are also tested by the algorithm. It has been observed that a relatively small network results when compared to that of cascor or caserr algorithms. The time complexity and computational expense is relatively large in this case. The best combination of aggregations and activations in a node, are observed with the partially trained network, which has different connections and their values; thereby different architectures result.

CONCLUSIONS

Summary

The area of dynamically altering neural network architecture is investigated. The advantage of constructive approach over pruning approach is studied. Constructive algorithms "Cascor", "Caserr", "Casall" and "Casany" are developed. These algorithms are validated on several benchmark problems. Generalized higher order neuron model is developed. The architectures that can be built by including different complexities, at both the node level and the layer level are explored and their performance is tested on the benchmark problems.

Cascading algorithms give results in steps. The change in step is relatively large for "Caserr", than that of "Cascor". The resultant architectures by "Cascor" and "Caserr" are nearer to each other. As there is a relatively large step change in "Caserr", it takes fewer epochs. "Casall" gives an architecture, which limits fan-in to the hidden units by building up layers, and is free from weight freezing. A less complicated structure, with fewer interconnections is resulted.

The impact of taking more than one aggregation function and activation function in a single unit is investigated. Generalized neuron model is developed. The network architectures that are homogeneous, heterogeneous and semi heterogeneous with same or different generalizations in the neuron are considered. The algorithm "Casany" that includes the best generalizations, during the build up process is developed. In case of

"casany", the time complexity increases for selecting the best topology for a hidden unit, and different topologies are resulted.

The algorithms developed are trained by different variants of back propagation and these methods are adapted to suit the learning process. The heuristics which bring in the best partially trained network are added to the algorithms. Improved variants of RPROP, with the simulated annealing(SARPROP) and re-annealing(ReSARPROP) terms are developed.

Scope for further work

Cascade algorithms overcome the limit of specifying the network architecture in advance. Near optimal architectures are results of these algorithms. In addition to dynamically increasing the network architecture, pruning of network connections can also be done, for developing optimal structures

In the generalized neuron model, the inputs can be enhanced by including mathematical functions like logarithmic, trigonometric and other functions.

Weight initialization forms an important aspect, which can be explored. Different topologies are resulted in case of "Casany", because of random initialization of weights. Further to reduce the time complexity of selecting the best topology in case of "Casany" algorithm, heuristics can be developed.

REFERENCES

[1] Waugh S, "Dynamic Learning Algorithms", Artificial Neural Network Research Group, Feb 1994.

[2] Scott E. Fahlman, Christian Lebiere, "The Cascade-Correlation Learning Architecture", Advances in Neural Information Processing Systems 2, D. S. Touretzky (ed.), Morgan Kaufmann Publishers, Los Altos CA, pp. 524-532, 1990.

[3] Mikko Lehtokangas, "Fast Initialization for Cascade-Correlation Learning", IEEE Trans. Neural Networks, Vol. 10, NO. 2, pp. 1335–1350, Mar. 1999.

[4] Nicholas K. Treadgold, Tam´as D. Gedeon, "Exploring Constructive Cascade Networks", IEEE Trans. Neural Networks, Vol. 10, NO. 6, pp. 1335–1350, Nov. 1999.

[5] Pathak D S, Koren I, "Connectivity and performance tradeoffs in the cascade correlation learning architecture", IEEE Trans. Neural Networks, Vol. 5, NO. 6, pp. 930–935, Nov. 1994.

[6] Nicholas K. Treadgold, Tam´as D. Gedeon, "Simulated annealing and weight decay in adaptive learning: The sarprop algorithm," IEEE Trans. Neural Networks, Vol. 9, pp. 662–668, 1998.

[7] Jang J.S.R., "ANFIS: Adaptive-Network-Based Fuzzy Inference System," IEEE Trans. Man and Cybernetics, Vol. 23,NO 23, pp. 665–685, 1993.

[8] J. M. Zurada, *Introduction to Artificial Neural Networks*, Delhi, Jaico Publishing House, 1994.

[9] M. H. Hassoun, *Fundamentals of Artificial Neural Networks*, New Delhi, Prentice Hall of India, 1998.

APPENDIX A

The Quickprop algorithm is as follows:

FOR each weight w_t

IF $\Delta w_{t-1} > 0$ **THEN**

IF $\frac{\partial E}{\partial w_t} > 0$ **THEN**

$$\Delta w_t = \eta * \frac{\partial E}{\partial w_t}$$

IF $\frac{\partial E}{\partial w_t} > \frac{\mu}{1+\mu} * \frac{\partial E}{\partial w_{t-1}}$ **THEN**

$$\Delta w_t = \Delta w_t + \mu * \Delta w_{t-1}$$

ELSE

$$\Delta w_t = \Delta w_t + \frac{\frac{\partial E}{\partial w_t}}{\frac{\partial E}{\partial w_{t-1}} - \frac{\partial E}{\partial w_t}} * \Delta w_{t-1}$$

ELSEIF $\Delta w_{t-1} < 0$ **THEN**

IF $\frac{\partial E}{\partial w_t} < 0$ **THEN**

$$\Delta w_t = \eta * \frac{\partial E}{\partial w_t}$$

IF $\frac{\partial E}{\partial w_t} < \frac{\mu}{1+\mu} * \frac{\partial E}{\partial w_{t-1}}$ **THEN**

$$\Delta w_t = \Delta w_t + \mu * \Delta w_{t-1}$$

ELSE

$$\Delta w_t = \Delta w_t + \frac{\frac{\partial E}{\partial w_t}}{\frac{\partial E}{\partial w_{t-1}} - \frac{\partial E}{\partial w_t}} * \Delta w_{t-1}$$

ELSE

$$\Delta w_t = \eta * \frac{\partial E}{\partial w_t}$$

$$w_t = w_t + \Delta w_t$$

APPENDIX B

The RPROP algorithm is as follows:

$\forall i, j : \Delta_{ij}(t) = \Delta_0$

$\forall i, j . \dfrac{\partial E}{\partial w_{ij}(t-1)} = 0$

REPEAT

Compute Gradient $\dfrac{\partial E}{\partial w_{ij}(t)}$

For all weights and biases

IF $\dfrac{\partial E}{\partial w_{ij}(t-1)} * \dfrac{\partial E}{\partial w_{ij}(t)} > 0$

$\Delta_{ij}(t)$=**Minimum** $\left(\Delta_{ij}(t-1) * \eta^{+}, \Delta_{\max}\right)$

$\Delta w_{ij}(t)$= **-sign** $\left(\dfrac{\partial E}{\partial w_{ij}(t)}\right) * \Delta_{ij}(t)$

$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$

$\dfrac{\partial E}{\partial w_{ij}(t-1)} = \dfrac{\partial E}{\partial w_{ij}(t)}$

ELSEIF $\dfrac{\partial E}{\partial w_{ij}(t-1)} * \dfrac{\partial E}{\partial w_{ij}(t)} < 0$

$\Delta_{ij}(t) = \Delta_{ij}(t-1) * \eta^{-}$

$\Delta_{ij}(t)$=**Maximum** $\left(\Delta_{ij}, \Delta_{\min}\right)$

$\dfrac{\partial E}{\partial w_{ij}(t-1)} = 0$

ELSEIF $\frac{\partial E}{\partial w_{ij}(t-1)} * \frac{\partial E}{\partial w_{ij}(t)} = 0$

$$\Delta w_{ij}(t) = \textbf{-sign}\left(\frac{\partial E}{\partial w_{ij}(t)}\right) * \Delta_{ij}(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

$$\frac{\partial E}{\partial w_{ij}(t-1)} = \frac{\partial E}{\partial w_{ij}(t)}$$

UNTIL CONVERGED

APPENDIX C

Electrical Load-Forecasting Problem

Electrical Load-Forecasting problem has also been considered, where the hourly load was predicted. To reflect the various factors affecting the hourly load (like temperature, moisture etc.), the inputs considered were,

1. The previous hour load (L-1)
2. Previous to previous hour load (L-2)
3 Previous day same hour load (L-24)
4. Previous day previous to previous hour load (L-25)
5. Previous week same hour load (L-168).

For training the system, only 176 patterns were presented, and 86 patterns were used for testing. The architecture used is one hidden layer of 3 neurons, which has SOP2 aggregation and sigmoid activation, the output neuron has summation term and sigmoid aggregation. The training results are shown in Fig C.1 and the test results are shown in Fig C 2.

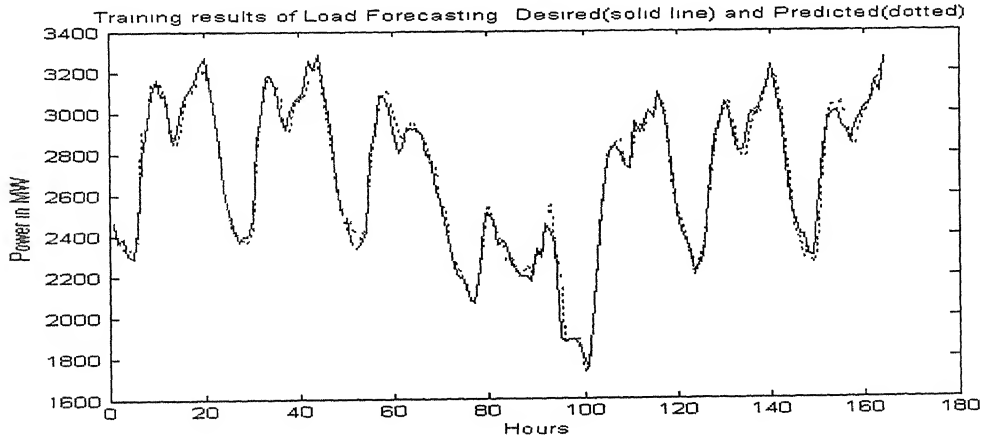


Fig C.1 Load Forecasting. Training results

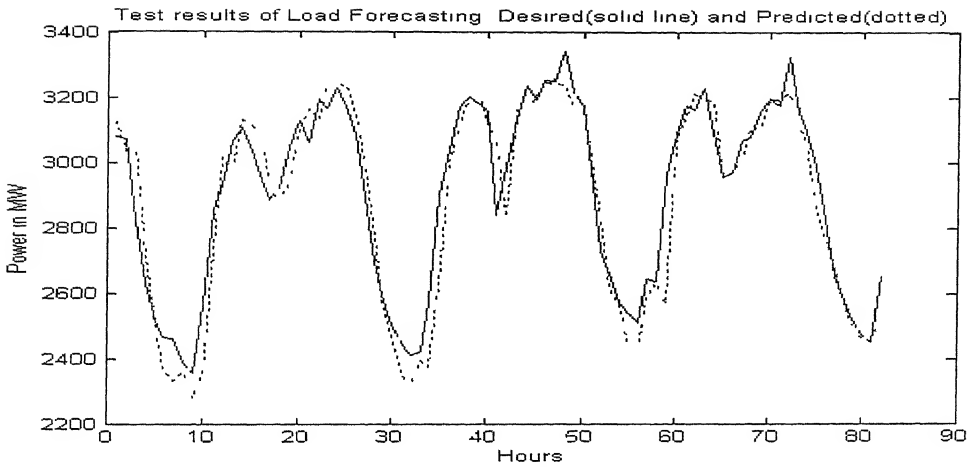


Fig C.2 Load Forecasting. Test results

The Function Approximation problem

A complicated multivariable function in seven dimensions has been selected for this problem. The function is as described below.

$$f(x_1, x_2, x_3, x_4, x_5, x_6) = \sum_{i=1,3,5} \sin(x_i)\sin(x_{i+1})\exp\left(-(x_i^2 + x_{i+1}^2)\right)$$

$$(-1 \leq x_1, x_2, x_3, x_4, x_5, x_6 \leq 1)$$

This six input non-linear function is trained by a single network with two hidden layers [8 8]. In the first hidden layer, all the neurons have summation term and sigmoid activation and in the second layer, all the neurons have SOP2 term and sigmoid activation. The output neuron is standard neuron. The results are plotted in figs C.3 and C.4.

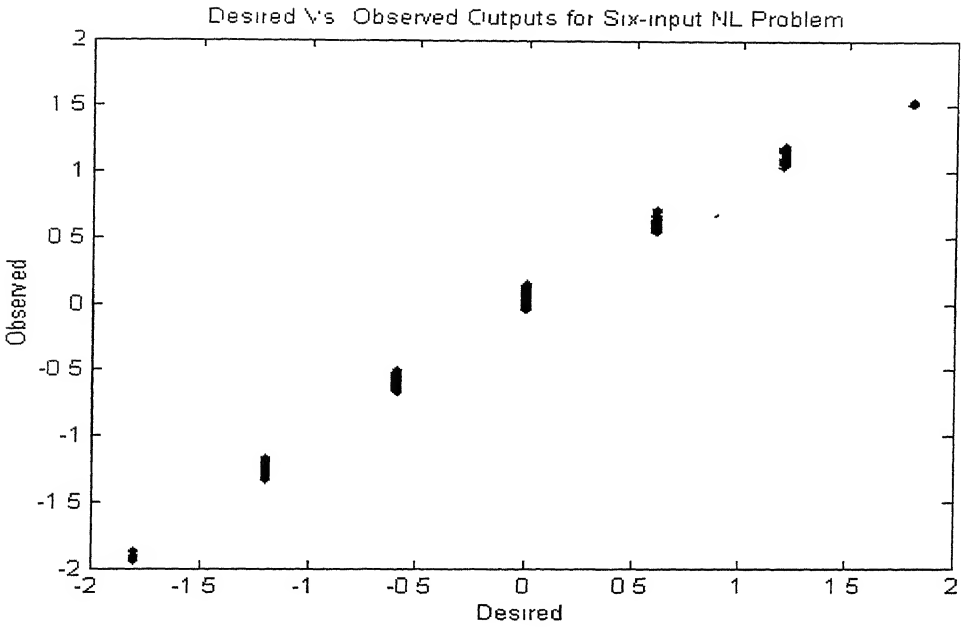


Fig C.3 Six-input NL problem. (Desired Vs Observed)

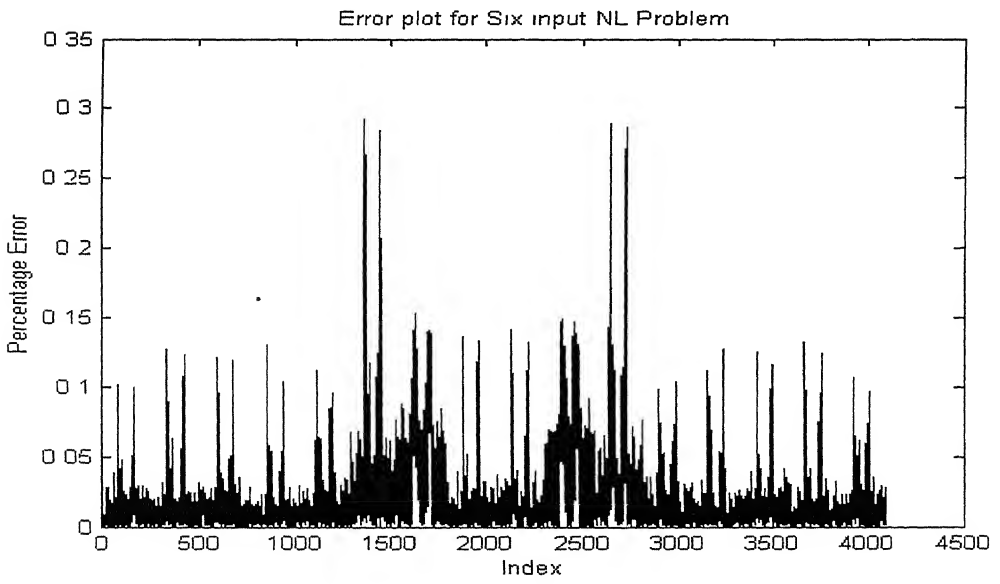


Fig C.4 Six-input NL problem. (Error at each pattern)

APPENDIX D

COMPARISON OF ARCHITECTURAL COMPLEXITY

Generalized Neural Network Vs Standard Neural Network

Sin(x)*Sin(y) Problem

Case	Architecture (Hidden)	Aggregations	Activations	Learning Method	Number of Connections
1	[6 4]	Summation, SOP2 And RB terms	Sigmoid, Tan Hyperbolic	BP-Batch	90
2	[10]	Summation, SOP2 And RB terms	Sigmoid, Tan Hyperbolic	SARPROP	120
3	[8 6]	Summation in the first layer, SOP2 in the second layer, Summation in the output layer	Sigmoid	BP-Batch, SARPROP	70

Table D.1 Results by Generalized Neural Network

The architectural complexity of standard neural network and generalized neural network for Sin(x)*Sin(y) problem is compared. Table D.1 shows the results for the case when higher order complexities are used. Table D 2 shows the results when Standard neural network is used (Reference: M.Tech Thesis " Comparison of performance of artificial neural network on typical benchmark problems using MATLAB tools vis-à-vis codes written in JAVA", by LT CDR SUSHIL KUMAR, 2001).

SinxSiny Problem: ANN Architectures used;				
Network	Architecture	Activation Fn.	Algorithm	Number of connections
SinxIny_n1	[15 15 15 1]	Tansig Tansıg Tansig Tansig	TrainSCG	541
SinxSıny_n11	[15 15 15 1]	Tansıg Tansig Tansig Logsig	TraınSCG	541
SinxSiny_n2	[25 25 1]	Tansıg Tansig Tansig	TrainSCG	751
SinxSiny_n21	[25 25 1]	Tansig Tansig logsig	TrainSCG	751
SinxSiny_n3	[20 25 30 1]	Tansig Tansig Tansig Logsig	TrrainRP	1396
SinxSiny_n4	[35 35 1]	Tansig Tansig Tansig	TrainRP	1401

Table D.2 Results by Standard Neural Network

From the comparison of results, it can be deduced that the inclusion of higher order terms ı e., by using generalized neuron as the basic node the number of weights required for solving a problem is greatly reduced.